Satunnaisten binääripuiden generointi

Jarmo Siltaneva

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos
Pro gradu -tutkielma
Elokuu 2000

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos
Jarmo Siltaneva: Satunnaisten binääripuiden generointi
Pro gradu -tutkielma, 99 sivua, 9 liitesivua

Elokuu 2000

Tässä tutkielmassa tutkitaan kuuden satunnaisia binääripuita generoivan algoritmin keskinäistä tehokkuutta erilaisten vertailukriteerien suhteen. Lisäksi näytetään, kuinka satunnaisia binääripuita generoivien algoritmien toteutusten oikeellisuus voidaan todentaa tilastollisesti. Tämän menetelmän avulla myös löydettiin virhe kirjallisuudessa esitetyssä Rémyn algoritmin esimerkkitoteutuksessa: kaikki samankokoiset puut eivät olleet yhtä todennäköisiä. Tutkielma sisältää myös satunnaisten puiden generointialgoritmien sekä binääripuiden luettelointialgoritmien bibliografiat. Lisäksi esitetään yksinomaan binääripuiden alkeisominaisuuksiin nojautuva todistus annetun asteen omaavien solmujen asymptoottiselle lukumäärän odotusarvolle satunnaisissa binääripuissa.

Avainsanat ja -sanonnat: puut, binääripuut, binääripuiden koodaukset, binääripuiden luettelointi, satunnaisten binääripuiden generointi.

Sisällys

1. Johdanto

Puut abstraktina rakenteena ovat yksi klassisista tutkimus- ja sovellusalueista tietojenkäsittelyopissa. Puita käytetään usein mallintamaan sellaisia ongelmia ja asioita, joiden loogista rakennetta voidaan luonnehtia mm. sanoilla haarautuva, hierarkkinen, osittava ja asteittain tarkentuva. Esimerkkeinä mainittakoon sukupuut, hajoita ja hallitse –algoritmin toimintaperiaate sekä kontekstittoman kieliopin lauseen tai aritmeettisen lausekkeen jäsennys.

Satunnaisten binääripuiden generointi (random binary tree generation, uniform random binary tree generation) on kombinatorisia rakenteita tutkivan algoritmi-tutkimuksen osa-alue. Ongelmanasettelu on yksinkertainen: kun puun solmujen lukumäärä on ennalta määrätty, generointialgoritmin on tuotettava puu (puita) sovitussa esitysmuodossa siten, että jokainen erilainen puu on yhtä todennäköinen. Oleellista on siis puun muoto, eivät puun solmujen mahdollisesti sisältämät tietoarvot.

Kombinatoristen rakenteiden satunnainen generointi on osa suurempaa ongelma-kokonaisuutta, jonka osaongelmat ovat sukua toisilleen. Olkoon $S=\{s_1,s_2,...,s_x\}$ määrätyn kokoisten ja tyyppisten kombinatoristen rakenteiden äärellinen joukko. Nämä osaongelmat ovat [Furnas, 1984] 1) joukon S alkioiden lukumäärän x selvittäminen, 2) joukon S jokaisen alkion esittäminen jossakin järjestyksessä ilman toistoja eli luettelointi (generation, listing), 3) joukon S mielivaltaisen alkion s_i järjestysnumeron i selvittäminen määrätyn järjestyksen sisällä (ranking), 4) joukon S alkion s_i konstruointi järjestysnumeron i perusteella (unranking) ja 5) joukon S satunnaisen alkion generointi (uniform random generation) siten, että jokaisen alkion todennäköisyys on 1/x.

Satunnaisten puiden tärkein sovellusalue on puita käsittelevien ohjelmien testaus ja analysointi: ohjelmalle syötteenä annettavia puita generoidaan satunnaisesti ja ohjelman oikeellisuutta tai suorituskykyä tutkitaan tilastollisin menetelmin. Toinen sovellusalue on abstraktien puiden graafinen esittäminen luonnollisten puiden kaltaisina.

Tämän tutkielman sisältö on seuraava. Luvussa 2 esitellään tutkielmassa käytettävät käsitteet, termit ja määritelmät sekä todistetaan eräitä keskeisiä binääripuiden ominaisuuksia. Lisäksi esitellään yleisimmät binääripuiden solmujen läpikäynti-järjestykset toteuttavat algoritmit. Luvun päättää erilaisten abstraktien puiden luokittelu.

Luvussa 3 kuvataan yksityiskohtaisesti tutkielmassa esiintyviä binääripuiden koodausmenetelmiä ja niiden ominaisuuksia.

Luku 4 on yleiskatsaus binääripuiden luettelointiin, joka on edellä listatuista

viidestä perusongelmasta kaikkein laajimmin tutkittu. Luvussa tarkastellaan binääripuiden keskinäisiin järjestyksiin ja luettelointialgoritmien tehokkuuteen liittyviä näkökohtia. Luvun päättää binääripuiden luettelointialgoritmien sekä numerointi- ja käänteisten numerointialgoritmien bibliografia.

Luvussa 5 kuvataan yksityiskohtaisesti satunnaisten binääripuiden generointialgoritmit sekä eräitä yleisempiä satunnaisten puiden generointialgoritmeja. Luvun lopussa on satunnaisten puiden generointialgoritmien sekä muutoin aihetta läheisesti sivuavien julkaisujen bibliografia.

Luvussa 6 esitetään tässä tutkielmassa käytettyihin käsitteisiin nojautuva todistus binääripuiden lukumäärän lausekkeelle sekä todistetaan eräitä satunnaisten binääripuiden asymptoottisia ominaisuuksia.

Luvuissa 7 ja 8 tutkitaan kuuden satunnaisia binääripuita generoivan algoritmin keskinäistä tehokkuutta erilaisten vertailukriteereiden avulla. Nämä luvut sisältävät tutkielman keskeiset tulokset. Algoritmien tehokkuuden tutkinta on ositettu seuraavasti: 1) satunnaisten binääripuiden koodausten generointi eli generointialgoritmien toteutusten kustannusten mittaus, 2) binääripuiden koodausten perusteella tapahtuva ‘lopullista’ muotoa olevien binääripuiden konstruointi eli konstruointialgoritmien toteutusten kustannusten mittaus sekä 3) tulosten analysointi. Lisäksi näytetään, kuinka generoitujen puiden satunnaisuutta voidaan tutkia tilastollisesti, jotta voidaan varmistua algoritmien toteutusten oikeellisuudesta.

2. Puut ja puiden määritelmät

Tässä luvussa esitellään käytettävät termit, käsitteet ja määritelmät. Suomenkielinen termistö noudattaa suurimmaksi osaksi Räihän [1988] oppikirjaa. Perinpohjainen esitys aiheeseen liittyvästä problematiikasta on Knuth [1973a].

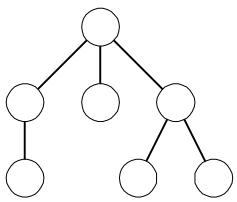
2.1. Puu

Määritelmä 2.1. *Puu* (tree, rooted tree) on

(i) tyhjä; tai

(ii) *juurisolmu* (root node), johon liittyy nolla tai useampia erillisiä *alipuita* (subtree), joista kukin on puu.

Puun atomaaristen alkioiden yleisnimitys on *solmu* (node, vertex). Juurisolmu eli juuri on siis puun solmu. Solmua kutsutaan juureksi yleensä vain silloin, kun se on koko tarkasteltavan puun (tai alipuun) määritelmänmukaisessa hierarkiassa ensimmäisenä. Alipuiden erillisyydellä tarkoitamme sitä, että saman juuren alipuilla ei voi olla yhteisiä solmuja. Esimerkki Määritelmän 2.1 mukaisesta puusta on kuvassa 1.



Kuva 1. Puu.

Jos solmun r alipuun juuri on solmu s, niin r on s:n *vanhempi* (parent, father) ja s on r:n *lapsi* (child, son). Vanhempisolmusta on tällöin yksisuuntainen yhteys lapsisolmuun, ja tätä yhteyttä kutsumme *särmäksi* (edge). Kahden särmällä yhdistetyn solmun *etäisyys* toisistaan on yksi. Solmun *aste* (degree) ilmoittaa solmun lasten lukumäärän. Jos solmun aste on nolla, solmu on *lehti* (leaf, external node), muussa tapauksessa *sisäsolmu* (branch node, internal node). Jos alipuiden järjestyksellä on merkitystä, puu on *järjestetty* (ordered). Tässä tutkielmassa kaikki puut ovat järjestettyjä, ellei toisin ole mainittu. Kutsumme puun *poluksi* (path) niitä solmuja, jotka läpikäymme alkaen puun juuresta edeten joka askeleella vanhemmasta lapseen ja päätyen johonkin lehtisolmuun, tässä järjestyksessä. Puun vasen (vast. oikea) polku

seuraa pelkästään vasempia (vast. oikeita) lapsia. Puun *korkeus* (height) on puun pisimmän polun solmujen lukumäärä.

Puun rakenne on luonnollista kuvata rekursiivisesti, kuten Määritelmässä 2.1 on tehty. Ei-rekursiivinen puun määrittely voidaan tehdä esim. käyttäen verkkoteorian termejä ja käsitteitä, abstraktin verkon erikoistapauksena (ks. kohta 5.11). Tätä tapaa käytetään paitsi verkkoteoreettisissa yhteyksissä usein myös diskreetin matematiikan oppikirjoissa, ks. esim. Rosen [1991].

Määritelmä 2.1 hyväksyy puuksi myös tapauksen, jossa solmujen lukumäärä on nolla. Käytäntö tässä vaihtelee: joskus puun määritelmissä edellytetään ainakin juuren olemassaolo [Knuth, 1973a; Aho and Ullman, 1995], joskus taas ei [Wirth, 1976; Weiss, 1994]. Edellisessä tapauksessa määritelmästä poistetaan ehto (i); puussa ei tällöin ole tyhjiä alipuita.

2.2. Binääripuu

Määritelmä 2.2. *Binääripuu* (binary tree) on

(i) tyhjä; tai

(ii) juurisolmu, johon liittyvät erilliset vasen alipuu ja oikea alipuu, jotka ovat binääripuita.

Käsitteet juuri, solmu, lehti, alipuu ja alipuiden erillisyys ovat analogisia puun kanssa. Kutsumme Määritelmän 2.2 mukaista puuta myös 'tavalliseksi' binääripuuksi.

Binääripuu kuvataan usein myös siten, että puun solmut jaetaan muodollisesti kahteen ryhmään, (sisä)solmuihin ja lehtiin. Tällöin määritelmän ehto (i) muuttuu:

Määritelmä 2.3. *Lehtisolmut sisältävä binääripuu* on

(i) lehti; tai

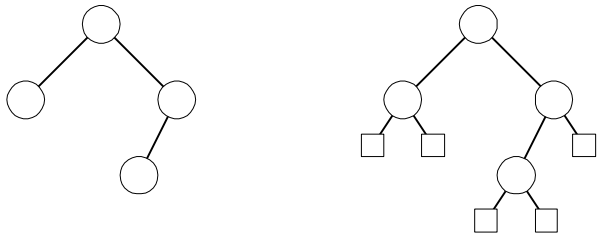
(ii) juurisolmu, johon liittyvät erilliset vasen alipuu ja oikea alipuu, jotka ovat lehtisolmut sisältäviä binääripuita.

Määritelmän 2.3 mukaista binääripuuta on kutsuttu kirjallisuudessa useilla eri nimillä: binääripuu; binääripuu, jossa on lehtisolmut (binary tree with external nodes) [Smith, 1989]; laajennettu binääripuu (extended binary tree) [Knuth, 1973a, p.399]; b-

puu (b-tree) [Knuth, 1973a, p.315]; kokonainen binääripuu (full binary tree) [Furnas, 1984].

Lemma 2.4. Määritelmien 2.2 ja 2.3 mukaisten puujoukkojen välillä on olemassa 1-1 -vastaavuus siten, että mielivaltaista tavallista binääripuuta T vastaava lehtisolmut sisältävä binääripuu S saadaan liittämällä T:n jokaisen tyhjän alipuun paikalle lehtisolmu.

Todistus. Jäsennetään puu T soveltaen toistuvasti Määritelmän 2.2 ehtoja. Rinnakkain konstruoidaan puu S soveltaen Määritelmän 2.3 vastaavasti numeroituja ehtoja samassa järjestyksessä. Määritelmien 2.2 ja 2.3 välisen analogian perusteella S on muutoin samanlainen kuin T, mutta jokaista T:n tyhjää alipuuta vastaa puussa S lehti. Menettely on yksikäsitteinen myös toiseen suuntaan, lähtien mielivaltaisesta lehtisolmut sisältävästä binääripuusta S päätyen tavalliseen binääripuuhun T. ?



Kuva 2. Binääripuu ja sitä vastaava lehtisolmut sisältävä binääripuu.

Kuvassa 2 on esimerkki Lemman 2.4 mukaisesta vastaavuudesta. Lemmaan 2.4 nojautuen esitämme todistuksen eräille tunnetuille binääripuiden ominaisuuksille.

Lause 2.5. Jos lehtisolmut sisältävässä binääripuussa on n sisäsolmua, niin siinä on n-1 särmää ja n+1 lehteä; vastaavasti n-solmuisessa tavallisessa binääripuussa on n-1 särmää ja n+1 tyhjää alipuuta; $n \geq 0$.

Todistus. Lemman 2.4 perusteella riittää, että johdamme vastaavan tavallisen binääripuun tyhjien alipuiden lukumäärän. Tavallisessa binääripuussa jokainen solmu sisältää tasan kaksi (mahdollisesti tyhjää) alipuuta. Solmujen lukumäärä on n, joten alipuiden lukumäärä on 2n. Ei-tyhjää alipuuta puussa vastaa särmä. Puun juuri poislukien jokaiseen solmuun tulee yksi särmä, ja se tulee solmun vanhempisolmusta. Särmien eli ei-tyhjien alipuiden lukumäärä on siis n-1. Tyhjien alipuiden lukumäärä on täten n+1, josta lause seuraa. ?

Erilaisten n-solmuisten binääripuiden lukumäärä on

$$C_n = \frac{1}{n+1}\binom{2n}{n},$$

joka tunnetaan yleisesti nimellä Catalanin luku, ja se esiintyy myös useiden muiden kombinatoristen ongelmien yhteydessä [esim. Gardner, 1976; Graham *et al.*, 1989; Sedgewick and Flajolet, 1996]. C_n kasvaa n:n suhteen eksponentiaalisesti, likimäärin kuten funktio $4^n/n^{1.5}$ [Knuth, 1973a]. Koska $C_{n+1}/C_n = (4n+2)/(n+2) = 4\text{-}6/(n+2)$, C_n:n arvot voidaan laskea ilman kertomafunktioita seuraavan rekursioyhtälön avulla:

$$C_0 = 1; \quad C_{n+1} = 4C_n - 6C_n/(n+2).$$

Binääripuiden lukumäärän lauseke on aihepiirimme kannalta keskeisessä asemassa, koska siihen perustuu mm. eräiden satunnaisia binääripuita generoivien algoritmien toiminta. Tästä syystä se todistetaan. Todistus esitetään kuitenkin vasta luvussa 6, koska tarvittavia käsitteitä ja apuvälineitä ei ole vielä esitelty.

Binääripuu on todennäköisesti kaikkein perusteellisimmin tutkittu abstrakti puu. Binääripuuta voitaneen pitää puiden ‘perustapauksena’ mm. seuraavista syistä: 1) Yleinen puu voidaan aina esittää binääripuun avulla, ja muunnos on riittävän yksinkertainen. Knuth [1973a] esitti tätä varten muunnossäännöt. 2) Solmujen aste on vakio. Silloin, kun solmujen aste on mielivaltainen, solmujen käsittelyyn tarvitaan dynaaminen tietorakenne, mikä tekee tarkasteluista monimutkaisempia. 3) Toteutettaessa puu, jonka solmujen maksimiaste on jokin vakio $k \geq 2$, binääripuissa minimoituvat osoitinmuuttujien lukumäärä ja täten myös tyhjän osoitintilan (nollaosoittimien) suhteellinen osuus puun koosta, ja puun muodolla ei ole vaikutusta [Horowitz and Sahni, 1987]. Jokainen solmu sisältää k osoitinmuuttujaa mahdollisia alipuita varten. Jos puun solmujen lukumäärä on n, osoitinmuuttujien lukumäärä koko puussa on nk. Näistä n-1 on käytössä, koska jokaiseen solmuun juurta lukuunottamatta tulee särmä vain solmun vanhempisolmusta. Näiden suhde $(n-1)/(nk)$ lähenee nollaa, kun k kasvaa.

2.3. Binääripuun solmujen keskinäiset järjestykset

Binääripuun solmujen joukossa voidaan määritellä järjestys, joka perustuu solmujen sijaintiin puussa. Jos puussa on n solmua, erilaisten järjestysten lukumäärä on periaatteessa n!, joista potentiaalisesti mielenkiintoisina ovat tietenkin vain sellaiset,

jotka voidaan kuvata siten, että kuvauksen koko ei riipu n:stä. Berztiss [1986] esitti luokittelun, joka seitsemään eri kategoriaan luokiteltuna kuvaa yhteensä 26 erilaista järjestystä ja näille periaatteellisen toteutuksen.

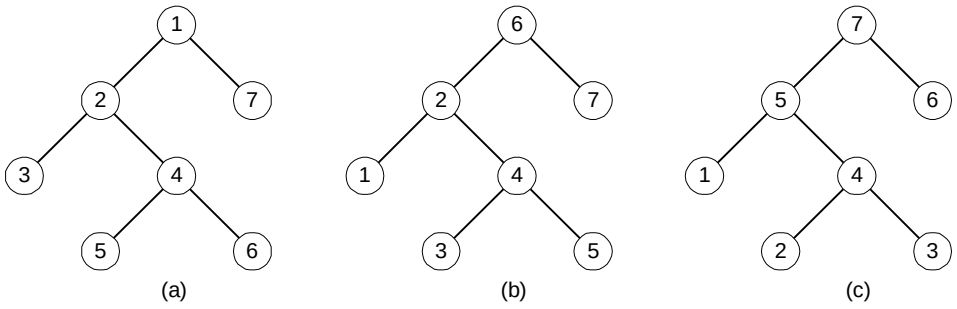
Kolme erilaista puun solmujen syvyyssuuntaiseen (depth-first) läpikäyntiin perustuvaa järjestystä ovat selvästi yleisimpiä. Ne ovat *esijärjestys* (preorder), *sisäjärjestys* eli *symmetrinen* järjestys (inorder, symmetric order) ja *jälkijärjestys* (postorder). Leveyssuuntaiseen (breadth-first) läpikäyntiin, vasemmalta oikealle, perustuva järjestys (level order) lienee seuraavaksi yleisin.

Esitetään seuraavassa algoritmit, jotka toteuttavat binääripuun läpikäynnin esijärjestyksessä (algoritmi 2.6) ja leveyssuuntaisessa järjestyksessä (algoritmi 2.7). Oletetaan binääripuu, jonka juurisolmu on r, ja lisäksi proseduuri Tulosta, joka tulostaa parametrina annetusta solmusta halutut tiedot. Nyt aliohjelmakutsu TraverseTree(r) tulostaa puun solmut esijärjestyksessä.

```
(1)  procedure TraverseTree(Solmu)
(2)  comment binääripuun solmujen tulostus esijärjestyksessä;
(3)  if (Solmu ei ole tyhjä) then
(4)      Tulosta(Solmu)
(5)      TraverseTree(Solmun vasen lapsi)
(6)      TraverseTree(Solmun oikea lapsi)
(7)  end if
(8)  end procedure
```

Algoritmi 2.6. Binääripuun solmujen läpikäynti/tulostus esijärjestyksessä.

Jos haluamme toteuttaa sisäjärjestyksen, siirrämme rivin 4 rivin 5 jälkeen. Jos haluamme toteuttaa jälkijärjestyksen, siirrämme rivin 4 rivin 6 jälkeen. Esi-, sisä- ja jälkijärjestyksessä vasemman alipuun solmut tulevat ennen oikean alipuun solmuja. Erot syntyvät siitä, mikä on juuren järjestys suhteessa sen alipuihin. Kuvassa 3 on esimerkkejä eri järjestyksistä.



Kuva 3. Binääripuun solmut numeroituna yhdestä seitsemään a) esijärjestyksessä, b) sisäjärjestyksessä ja c) jälkijärjestyksessä.

Syvyyssuuntaiseen läpikäyntiin perustuvat algoritmit heijastavat puun rakennetta ja ovat rekursiivisessa muodossaan yksinkertaisia ja selkeitä. Leveyssuuntaisen läpikäynnin (algoritmi 2.7) toteutus on yksinkertaisinta jonon avulla, joka on FIFO-periaatteella (first-in-first-out) toimiva standarditietorakenne. Algoritmi tulostaa vasemmalta oikealle ensin puun juuren, sitten juuren lapset, sitten juuren lastenlapset ja niin edelleen. Jono-operaatioiden nimet olkoot VieJonoon ja OtaJonosta, ja muut nimet ja oletukset ovat samat kuin edellä.

```
procedure TraverseTree(Solmu)
comment binääripuun solmujen tulostus leveyssuuntaisessa järjestyksessä;
VieJonoon(Solmu)
while (Jono ei ole tyhjä) do
   OtaJonosta(Solmu)
   Tulosta(Solmu)
   if  (Solmulla on vasen lapsi) then VieJonoon(Solmun vasen lapsi) end if
   if  (Solmulla on oikea lapsi)  then VieJonoon(Solmun oikea lapsi) end if
end while
end procedure
```

Algoritmi 2.7. Binääripuun solmujen läpikäynti/tulostus leveyssuuntaisessa järjestyksessä.

2.4. Puiden luokittelu

Tämän luvun lopuksi esitämme erilaisten kirjallisuudessa esiintyvien puiden luokittelun niiden ominaisuuksien perusteella [Furnas, 1984]:

1) Puun solmujen aste d

   -on mielivaltainen; tämä on Määritelmän 2.1 mukainen puu

   -voi vaihdella välillä $1 \leq d \leq t$ (t-ary tree); tällaisesta puusta käytämme jatkossa nimitystä *t-puu*; kun t=2, kyseessä on Määritelmän 2.2 mukainen binääripuu

   -voi olla joko 0 tai t (full t-ary tree); tämä on lehtisolmut sisältävä t-puu; kun t=2, kyseessä on Määritelmän 2.3 mukainen binääripuu

2) Solmujen alipuiden järjestys

   - ei ole määritelty (unordered tree, oriented tree)

   - on määritelty (ordered tree, plane tree)

3) Solmujen numerointi

- ei numerointia (unlabeled tree)
- vain lehtisolmut numeroidaan (terminally labeled tree)
- kaikki solmut numeroidaan (labeled tree, completely labeled tree)

4) Puun juuri

- ei ole määritelty (unrooted tree, free tree); tällaisesta puusta käytämme jatkossa nimitystä *vapaa puu*
- on määritelty (rooted tree).

Ylläolevan listan näkökulma on kombinatorinen, ts. puita ei varsinaisesti tarkastella tietojen tallettamisen apuvälineinä. Kokonaisuutena ottaen kirjallisuudessa esiintyviä erilaisia puita on suuri määrä [ks. esim. Gonnet ja Baeza-Yates, 1991]. Useimmille variaatioille yhteistä on pyrkimys erilaisin tavoin rajoittaa puun korkeutta ja tasapainottaa puun muotoa solmujen lisäys- ja poisto-operaatioiden yhteydessä.

3. Binääripuiden koodaukset

Binääripuun kaksiulotteinen rakenne on mahdollista kuvata yksiulotteisessa muodossa sovitun *koodauksen* eli *koodisanan* (codeword) avulla. Tässä luvussa esitellään tutkielmassa tarvittavat binääripuiden koodaukset, sikäli kuin niitä ei esitellä generointialgoritmien yhteydessä. Kaikissa tarkasteltavissa menetelmissä puu esitetään kokonaislukulistoina tai merkkijonoina. Luku perustuu pääasiassa Mäkisen [1991] yhteenvetoon algoritmitutkimuksessa esiintyvistä binääripuiden koodausmenetelmistä ja siinä esitettyihin viitteisiin.

3.1. Koodausmenetelmien luokittelu

Koodausta käytetään usein sellaisissa kombinatorisissa puualgoritmeissa, joissa oleellinen asia on puun muoto eivätkä esim. solmujen mahdollisesti sisältämät tietoarvot. Ongelman ratkaisutapa yleensä muuttaa muotoaan aivan toiseksi, kun ongelma ratkaistaan alkuperäistä kombinatorista rakennetta vastaavaan koodaukseen perustuen (eikä esim. tietue- ja osoitinmuuttujien ja niihin kohdistuvien operaatioiden avulla). Näin pyritään saavuttamaan jokin etu, esim. ongelman monimutkaisuuden väheneminen, resurssien käytön väheneminen, algoritmien käyttämien lukujen pieneneminen tai algoritmin kuvaamisen yksinkertaistuminen.

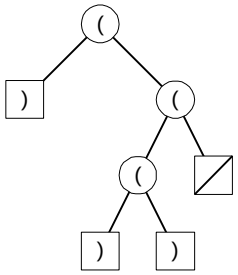
Puuta edustavien koodisanojen on täytettävä seuraavat vaatimukset: 1) Koodisanan on oltava yksikäsitteinen eli puuta vastaa täsmälleen yksi koodisana ja koodisanaa vastaa täsmälleen yksi puu. Kuvaus puiden joukolta koodisanojen joukolle on siis bijektio. 2) Koodisanan perusteella on riittävän vähin kustannuksin pystyttävä konstruoimaan sitä vastaava binääripuu ja päinvastoin. Kaikissa tarkastelemissamme koodausmenetelmissä lopullista muotoa oleva binääripuu voidaan konstruoida koodauksen perusteella lineaarisessa ajassa ja tilassa.

Erilaisia puiden koodauksia voidaan luokitella seuraavasti: 1) Menetelmät, joissa puun sisäsolmuihin ja/tai lehtiin liitetään jokin tunniste tai arvo, yleensä kokonaisluku. Koodisanaksi saadaan tunnistelista, jonka alkioiden järjestys määräytyy solmujen läpikäyntijärjestyksen perusteella. Tähän kaavaan sopivat menetelmät ovat moninaiset, ja kutsumme niitä ‘tavallisiksi’ koodausmenetelmiksi. 2) Permutaatioihin perustuvat koodaukset. Tarvitaan kaksi puun läpikäyntiä: ensin numeroidaan puun solmut 1:stä n:ään valitussa solmujen läpikäyntijärjestyksessä, jonka jälkeen numerot luetaan

toisessa järjestyksessä. Koodisanaksi saadaan täten lukujen 1...n permutaatio. 3) Rotaatioihin perustuvat koodaukset. Rotaatio, joka kuvataan tarkemmin jäljempänä, on puun rakennetta muuttava operaatio, joka ‘kierrättää’ yhden tai useampia solmuja oikealle tai vasemmalle. Rotaatioon perustuvissa koodausmenetelmissä lasketaan valitussa järjestyksessä kullekin solmulle tarvittava rotaatio-operaatioiden lukumäärä, jotta saavutetaan jokin sovittu puun muoto (esim. vasen lista eli puu, jossa kaikki oikeat alipuut ovat tyhjiä). Koodisana koostuu listasta rotaatio-operaatioiden lukumääriä. 4) Muut menetelmät, joita ei tässä esityksessä käsitellä.

3.2. Tavalliset koodausmenetelmät

Tasapainotettu (balanced, well-formed) *sulkujono* koostuu alku- ja loppusuluista ja täyttää seuraavat ehdot: 1) alkusulkujen lukumäärä on sama kuin loppusulkujen lukumäärä; 2) tutkittaessa jonoa merkki kerrallaan vasemmalta oikealle, loppusulkujen lukumäärä ei koskaan ylitä alkusulkujen lukumäärää.



Kuva 4. Tasapainotettua sulkujonoa ()(()) vastaava binääripuu.

Tasapainotettujen sulkujonojen ja binääripuiden välillä on tunnettu 1-1 -vastaavuus [esim. Zaks, 1980; Er, 1983]. Olkoon T mielivaltainen lehtisolmut sisältävä binääripuu. Puussa on tällöin n solmua ja n+1 lehteä; n≥0. Puu läpikäydään esijärjestyksessä. Kun tavataan solmu, tulostetaan alkusulku, ja kun tavataan lehti, tulostetaan loppusulku (kuva 4). Syntyvän sulkujonon pituus on 2n+1. Koska viimeisenä läpikäyty puun alkio on aina välttämättä lehti, voimme poistaa jonon viimeisen alkion eli loppusulun. Lopputuloksena saadun jonon pituus on 2n, joten ehto 1) seuraa. Esijärjestyksen määritelmän mukaan (ks. algoritmi 2.6) jokainen solmu läpikäydään ennen kuin sen lapsi, joka mahdollisesti on lehti, joten ehto 2) seuraa. Menettely on käännettävissä lähtien mielivaltaisesta ehdot 1) ja 2) täyttävästä sulkujonosta päätyen yksikäsitteiseen

lehtisolmut sisältävään binääripuuhun; tämän toteuttava algoritmi esitetään kohdassa 7.4.1.

Zaksin [1980] käyttämä binääripuiden koodaus (Zaksin jono) on sama kuin yllä, mutta kukin alkusulku korvataan ykkösellä ja kukin loppusulku nollalla.

Leen *et al*. [1986] koodauksessa binääripuun solmut merkitään samoin kuin Zaksin jonoissa, mutta koodisana muodostetaan lukemalla puun solmut leveyssuuntaisessa järjestyksessä (ks. algoritmi 2.7). Viimeinen alkio on aina nolla, ja se jätetään huomioimatta. Myöskin tämä koodisana täyttää tasapainotetun sulkujonon ehdot, mutta se (yleensä) edustaa eri puuta kuin Zaksin jono.

Olkoon binääripuun juuren *taso* (level) 0, juuren lasten taso 1, juuren lastenlasten taso 2 jne. *Tasojono* (level sequence) muodostetaan läpikäymällä puun n solmua ja n+1 lehteä symmetrisessä järjestyksessä siten, että joko 1) vain kunkin sisäsolmun kohdalla listataan sen taso, tai 2) vain kunkin lehden kohdalla listataan sen taso. Edellisessä tapauksessa koodisanan pituudeksi tulee n, jälkimmäisessä n+1 [Ruskey and Hu, 1977].

Painojono (weight sequence) muodostetaan liittämällä (lehtisolmut sisältämättömän) binääripuun jokaiseen solmuun sen vasemman alipuun solmujen lukumäärä lisättynä yhdellä. Koodisana generoidaan tulostamalla nämä lukumäärät symmetrisessä järjestyksessä [Pallo, 1986]. Samanarvoinen koodausmenetelmä on laskea solmujen oikeiden alipuiden lukumäärät vastaavasti.

3.3. Puupermutaatiot

Puuta edustava koodisana voidaan muodostaa seuraavasti: läpikäydään puun solmut symmetrisessä järjestyksessä samalla numeroiden ne ykkösestä alkaen. Sen jälkeen läpikäydään puun solmut esijärjestyksessä samalla listaten solmujen numerot. Tuloksena saatu kokonaislukulista on jokin lukujen 1...n permutaatio, jota kutsutaan *puupermutaatioksi* (tree permutation) [Knott, 1977]. Kuvan 3 binääripuuta vastaava puupermutaatio on 6214357.

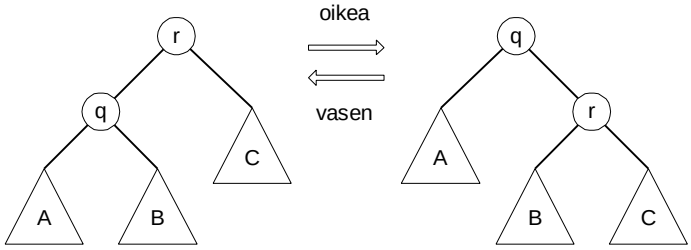
Puupermutaatio on aina muotoa p_1p'p'', missä merkinnät ovat seuraavat: p_1 on puun juuren numero; p' on vasemman alipuun puupermutaatio eli lukujen 1...p_1-1 permutaatio; p'' on oikean alipuun puupermutaatio luku p_1 lisättynä jokaiseen alkioon eli lukujen p_1+1...n permutaatio. Lukujen 1...n permutaatio p on puupermutaatio silloin ja vain silloin kun se ei sisällä sellaista alijonoa $p_i p_j p_k$, että $p_k < p_i < p_j$ [Rotem, 1975; Knott, 1977].

Puu voidaan konstruoida puupermutaation p_1p'p'' perusteella rekursiivisesti seuraavalla tavalla: jos puupermutaatio on tyhjä, puu on valmis; muulloin 1) luo puun juuri; 2) luo vasen alipuu puupermutaation p' perusteella; 3) luo oikea alipuu puupermutaation p'' perusteella [Knott, 1977; Er, 1987; vrt. kohta 5.13].

Millaiset kaksi puun läpikäyntiä määrittelevät yksikäsitteisesti puuta edustavan koodisanan? Käytetään seuraavia merkintöjä: h on puun juuri, L vasen alipuu ja R oikea alipuu. Merkitköön hLR puun läpikäyntiä esijärjestyksessä, LhR symmetrisessä järjestyksessä ja LRh jälkijärjestyksessä. Tämän kohdan puupermutaatiot saatiin läpikäyntiparin (LhR, hLR) tuloksena. On olemassa neljä olennaisesti erilaista läpikäyntijärjestysparia, jotka määrittelevät yksikäsitteisen 1-1 -vastaavuuden binääripuiden ja koodisanojen välille: (LhR, LRh), (LhR, RLh), (LhR, hLR), (LhR, hRL) [Hille, 1985].

3.4. Rotaatioihin perustuvat koodaukset

Rotaatio (rotation) on operaatio, joka muuttaa binääripuun muotoa siten, että solmujen symmetrinen järjestys pysyy muuttumattomana. Olkoon solmun r vasen lapsi solmu q. Oikea(npuoleinen) rotaatio r:n ja q:n välillä siirtää 1) solmun r solmun q oikeaksi lapseksi, 2) mahdollisen entisen q:n oikean lapsen r:n vasemmaksi lapseksi ja 3) mahdollisen entisen r:n vanhempisolmun q:n vanhempisolmuksi. Jos tuloksena saadussa puussa tehdään vasen rotaatio q:n ja r:n välillä, puu palautuu alkuperäiseen muotoonsa (kuva 5). Binääripuiden rotaatioista tarkemmin ks. esim. Knuth [1973b] tai Weiss [1994].



Kuva 5. Oikea ja vasen rotaatio solmujen r ja q välillä.

Zerlingin [1985] koodausmenetelmässä kuvittelemme puun solmut merkityksi numeroilla 1...n symmetrisessä järjestyksessä. Koodisana määräytyy seuraavasti: suoritamme vasempia rotaatioita puun juuren kohdalla niin kauan kuin juuren oikea

lapsi ei ole tyhjä. Kun rotaatioita ei enää voi suorittaa, puun juuren kohdalla on solmu n. Nyt koodisanan *viimeinen* alkio saa arvokseen suoritettujen rotaatioiden lukumäärän. Tämän jälkeen poistamme solmun n ja suoritamme vastaavat toimenpiteet uuden juuren kohdalla, ja niin edelleen. Lopulta puussa on vain solmu 1, joten koodisanan ensimmäiseksi alkioksi tulee aina 0. Huomattakoon, että solmujen numerointi ja juuren poisto on tässä tehty havainnollisuussyistä: algoritmin toiminnan kannalta ne ovat tarpeettomia. Puu on lopputilanteessa vasen lista. Puun konstruointi koodisanan perusteella tapahtuu suorittamalla edelläkuvatut toimenpiteet päinvastaisessa järjestyksessä suorittaen oikeanpuoleisia rotaatioita.

Vasemman etäisyyden menetelmässä [Mäkinen, 1987] suoritamme vasempia rotaatioita solmun n ja sen kulloisenkin vanhempisolmun välillä, kunnes solmu n on puun juuressa. Koodisanan n:s alkio saa arvokseen suoritettujen rotaatioiden lukumäärän. Poistamme juuren, ja suoritamme vastaavat toimenpiteet kohdistuen solmuun n-1 vastaavasti, ja niin edelleen. Koska valmiin koodisanan kunkin alkion x_i arvoksi tulee solmun i etäisyys puun vasemmasta polusta alkutilanteessa [Mäkinen, 1987], pelkästään koodisanan selville saamiseksi rotaatioita ei itse asiassa tarvitse suorittaa. Solmujen numerointia ja juuren poistoa koskevat huomautukset ovat samat kuin Zerlingin menetelmän kohdalla.

4. Binääripuiden luettelointi

Tässä luvussa tarkastellaan binääripuiden luettelointialgoritmeja sekä puiden keskinäisten järjestysten määrittelyyn ja luettelointialgoritmien tehokkuuteen liittyviä näkökohtia. Algoritmien sisäistä toimintaa ei yksityiskohtaisesti tarkastella. Luvun päättää luettelointialgoritmien bibliografia.

4.1. Binääripuiden keskinäiset järjestykset

Binääripuiden joukossa voidaan määritellä järjestys, joka perustuu puun muotoon, kun solmujen lukumäärä on n. Erilaisten järjestysten lukumäärä on tällöin periaatteessa C_n!, joista potentiaalisesti mielenkiintoisia ovat tietenkin vain sellaiset, jotka voidaan kuvata siten, että kuvauksen koko ei riipu n:stä. Binääripuiden järjestyksiin liittyvät perusoperaatiot (vrt. johdantoluku) ovat seuraavat:

- annetun binääripuun järjestysnumeron selvittäminen valitun järjestyksen sisällä eli numerointi (ranking); tulos voi olla välillä 1...C_n
- annettua järjestysnumeroa vastaavan binääripuun selvittäminen (unranking); käänteinen toimenpide edelliselle
- kaikkien binääripuiden peräkkäinen generoiminen yksi kerrallaan ensimmäisestä viimeiseen jossakin järjestyksessä eli *luettelointi* (generation, listing).

Useimmat edellämainittuja perusoperaatioita suorittavat algoritmit perustuvat puita edustavien koodisanojen manipulointiin. Koska koodisanan perusteella lopullinen puu on aina yksikäsitteisesti konstruoitavissa, termillä ‘puu’ voidaan viitata yhtä hyvin abstraktiin puuhun kuin sitä vastaavaan koodisanaankin.

Järjestysrelaatio ‘<‘ (pienempi kuin) kahden mielivaltaisen puun välillä automaattisesti määrittelee järjestyksen koko puiden joukossa. Knott [1977] määritteli järjestysrelaation seuraavasti:

Määritelmä 4.1. Mielivaltaisille binääripuille T ja S pätee T<S, jos

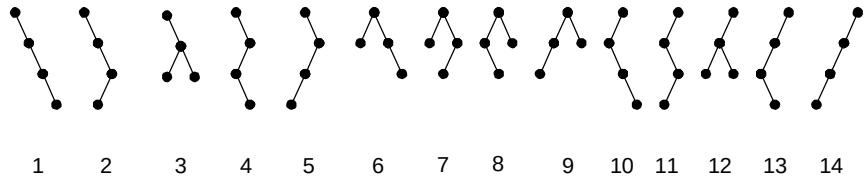
(i) size(T)<size(S); tai

(ii) size(T)=size(S) ja left(T)<left(S); tai

(iii) size(T)=size(S) ja left(T)=left(S) ja right(T)<right(S),

missä size() on puun solmujen lukumäärä ja left() ja right() ovat puun juuren vasen ja oikea alipuu; ehdossa (iii) esiintyvän kahden binääripuun välisen yhtäsuuruusrelaation ’=’ oletamme tunnetuksi.

Määritelmä 4.1 oli muunnelma Knuthin [1973a, Ex. 2.3.1-25, 2.3.2-8] esittämästä järjestysrelaatiosta, joka perustui osittain myös puun solmuissa olevien avaintietojen käyttöön. Määritelmän 4.1 mukaista järjestystä kutsutaan yleisesti ‘luonnolliseksi’ tai ‘globaaliksi’ järjestykseksi binääripuiden joukossa. Esimerkki tästä järjestyksestä on kuvassa 6. Knott esitti ko. järjestyksen toteuttavan numerointi- ja vastaavan käänteisalgoritmin, muttei varsinaista luettelointialgoritmia.



Kuva 6. Nelisolmuiset binääripuut Määritelmän 4.1 mukaisessa järjestyksessä [Knott, 1977].

Toinen yleisesti esiintyvä järjestys määritellään seuraavasti:

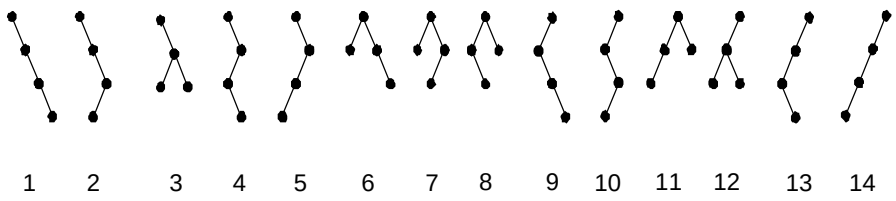
Määritelmä 4.2. Mielivaltaisille binääripuille T ja S pätee T<S, jos

(i) T on tyhjä ja S ei ole tyhjä; tai

(ii) T ja S eivät ole tyhjiä ja left(T)<left(S); tai

(iii) T ja S eivät ole tyhjiä ja left(T)=left(S) ja right(T)<right(S).

Määritelman 4.2 mukaista järjestystä kutsutaan yleisesti ‘paikalliseksi’ järjestykseksi. Esimerkki tästä järjestyksestä on kuvassa 7. Ero globaaliin järjestykseen



Kuva 7. Nelisolmuiset binääripuut Määritelmän 4.2 mukaisessa järjestyksessä [Zaks, 1980].

voidaan ilmaista vertailemalla päättelysääntöjä molemmissa menetelmissä. Tutkittaessa, onko T<S, läpikäydään puun solmuja esijärjestyksessä. Kun sovelletaan globaalia järjestystä, tutkitaan rinnakkain molempien puiden solmujen alipuiden kokoja, ts. puuta koskevaa globaalia informaatiota, toisin kuin paikallisen järjestyksen kyseessä ollessa [Zaks, 1980]. Paikallisen järjestyksen toteuttaminen on tästä syystä yksinkertaisempaa [esim. Er, 1987].

4.2. Luettelointialgoritmit

Binääripuiden luettelointi voidaan suorittaa tekemällä käänteinen numerointioperaatio järjestyksessä kaikille kokonaisluvuille 1...C_n. Varsinaiset luettelointialgoritmit kuitenkin tuottavat annetun binääripuun perusteella järjestyksessä sitä seuraavan binääripuun, mikä on tehokkaampi tapa. Jokainen luettelointialgoritmi automaattisesti määrittelee binääripuiden joukossa järjestyksen.

Ensimmäisenä julkaistun binääripuiden luettelointialgoritmin esittivät Ruskey ja Hu [1977] käyttäen koodauksena tasojonoja. Tasapainotettuja sulkujonoja generoivan luettelointialgoritmin esitti Zaks [1980], joka lisäksi osoitti, että molemmat algoritmit generoivat puita paikallisen järjestyksen mukaisesti.

Ruskeyn ja Hun sekä Zaksin algoritmeissa keskimääräinen aikavaatimus järjestyksessä seuraavan puun generointioperaatiolle on O(1), mutta huonoimmassa tapauksessa yksi operaatio tarvitsee ajan O(n). Parhaatkin sen jälkeen julkaistut algoritmit olivat niinikään keskimäärin O(1), mutta D. Roelants van Baronaigien [1991] esitti ensimmäisenä binääripuiden luettelointialgoritmin, joka ei sisällä lainkaan silmukoita (eikä rekursiota): tämä takaa aikavaatimuksen O(1) huonoimmassakin tapauksessa. Käytetty koodausmenetelmä perustui Zerlingin [1985] menetelmään. Ensimmäiset silmukattomat luettelointialgoritmit tasapainotetuille sulkujonoille esittivät toisistaan riippumatta Mikawa ja Takaoka, Vajnovszki sekä Walsh [1998] (ks. Vajnovszki [1998]).

Luettelointialgoritmit on yleensä pyritty toteuttamaan niin, että tehokkuusvaatimusten lisäksi jokin seuraavista kriteereistä täyttyy:

1) Puita edustavat koodisanat generoidaan suuruusjärjestyksessä (lexicographic order). Esimerkki tällaisesta on algoritmi 4.3.

2) Koodisanat generoidaan sellaisessa järjestyksessä, että jokaisen kahden

peräkkäisen koodisanan ero minimoituu, kun kriteereinä käytetään toisistaan eroavien merkkien lukumäärää ja sijaintia. Tällöin koodisanojen joukon ja luettelointi-järjestyksen sanotaan muodostavan (kombinatorisen, binääripuita koskevan) *Gray-koodin* (kombinatorisista Gray-koodeista ks. esim. Joichi *et al.* [1980], Reingold *et al.* [1977]). Proskurowski ja Ruskey [1985] esittivät algoritmin, joka generoi Gray-koodin Zaksin jonoille; tehokkaamman ja yksinkertaisemman algoritmin esittivät Ruskey ja Proskurowski [1990].

3) Koodisanat (tai puut) generoidaan sellaisessa järjestyksessä, että jokaisen kahden peräkkäisen lopullista muotoa olevan binääripuun rakenteellinen ero minimoituu. Tämä ero voidaan kvantifioida esim. *rotaatioetäisyyden* (rotation distance) avulla: montako rotaatiota tarvitaan, jotta jompikumpi puu voidaan muuntaa samanmuotoiseksi kuin toinen. Esimerkiksi Knottin [1977], Ruskeyn ja Hun [1977], Rotemin ja Varolin [1978], Zaksin [1980] ja Zerlingin [1985] algoritmien määrittelemissä järjestyksissä on olemassa kaksi peräkkäistä puuta, joiden rotaatioetäisyys on O(n), kun taas Proskurowskin ja Ruskeyn [1985] järjestyksessä kahden peräkkäisen puun rotaatio-etäisyys on aina joko 1 tai 3 [Lucas, 1987]. Lucasin [1987] luettelointialgoritmissa vastaava etäisyys on aina 1.

Luettelointialgoritmien yhtäläisyyksiä ovat tutkineet Lucas *et al*. [1993], jotka mm. osoittivat, että huolimatta eri koodausmenetelmistä seuraavissa luettelointialgoritmeissa samankokoisten binääripuiden luettelointiprosessin synnyttämät dynaamiset implisiit-tiset rekursiopuut ovat samoja: Zerling [1985], Pallo [1986], Mäkinen [1987]. Kielioppipohjaisen luettelointialgoritmin esittivät Xiang *et al.* [1997], jotka vertailivat sen tehokkuutta Zerlingin [1985], Pallon [1986] ja Erin [1985] algoritmeihin käyttäen kriteerinä suoritettujen rekursiivisten kutsujen lukumäärää. Yleisemmin kombina-toristen rakenteiden luettelointia on tutkinut Kemp [1998], joka mm. näytti, kuinka sopivaan koodaukseen nojautuen monet tunnetut luettelointialgoritmit noudattavat samaa yleistä peruskaavaa: tulos pätee mm. annetun kokoisille permutaatioille, mielivaltaisille säännöllisen kielen sanoille, joukon osajoukoille sekä binääripuille Ruskeyn ja Hun [1977] ja Zaksin [1980] mukaan.

Esimerkkialgoritmi 4.3 [Er, 1983] listaa kaikki tasapainotetut sulkujonot suuruus-järjestyksessä (ks. myös Zaks [1980]; Semba [1981]; Er [1985]). Algoritmin toteuttama binääripuiden järjestys on Määritelmän 4.2 mukainen. Sulkujonot tuotetaan globaaliin

taulukkoon p, jonka koko on 2n. Proseduuri generoi L alkusulkua ja L+R loppusulkua suuruusjärjestyksessä. Kutsu on muotoa GenParentheses(n,0).

```
procedure GenParentheses(L,R)
comment  Er [1983]: tasapainotettujen sulkujonojen luettelointi
if  L=0 and R=0
then     Tulosta(p)
else     i := 2*(n-L)-R+1
         if  R≠0 then  p[i] := LoppuSulku;  GenParentheses(L,R-1)     end if
         if  L≠0 then  p[i] := AlkuSulku;    GenParentheses(L-1,R+1) end if
end if
end procedure
```

Algoritmi 4.3. Tasapainotettujen sulkujonojen luettelointi suuruusjärjestyksessä.

4.3. Bibliografia

Binääri- ja muunlaisten puiden luettelointialgoritmeja on 1970-luvun lopulta lähtien julkaistu monia. Erilaisia koodausmenetelmiä ja järjestyksiä soveltaen binääripuiden (tai jos erikseen mainittu, muunlaisten puiden) luettelointialgoritmeja ovat esittäneet mm. seuraavat, aikajärjestyksessä listattuna:

- Ruskey ja Hu [1977] (tasojonot)
- Rotem ja Varol [1978] (arpajonot (ballot sequences))
- Ruskey [1978] (t-puu, Ruskeyn ja Hun [1977] yleistys)
- Trojanowski [1978] (t-puu, puupermutaatioihin perustuva koodaus)
- Zaks ja Richards [1979] (t-puun muunnelma)
- Beyer ja Hedetniemi [1980] (yleinen puu, Ruskeyn [1978] yleistys)
- Proskurowski [1980] (Zaksin jonot muunnelmineen)
- Solomon ja Finkel [1980] (ei koodausta)
- Zaks [1980] (Zaksin jonot, t-puu)
- Semba [1981] (tasapainotetut sulkujonot)
- Er [1983] (tasapainotetut sulkujonot)
- Hikita [1983] (k-kokoisten alipuiden luettelointi)
- Er [1985] (tasapainotetut sulkujonot)
- Pallo ja Racca [1985] (P-jonot, arpajonojen muunnelma; L-jonot, solmujen painoihin perustuva koodaus)
- Proskurowski ja Ruskey [1985] (tasapainotetut sulkujonot, Gray-koodi)

- Zerling [1985] (rotaatioihin perustuva koodaus)
- Lee *et al.* [1986] (binääripuut, joiden maksimikorkeus on annettu)
- Pallo [1986] (painojonot)
- Er [1987] (puupermutaatiot)
- Lucas [1987] (rotaatioihin perustuva koodaus)
- Mäkinen [1987] (rotaatioihin perustuva koodaus)
- Roelants van Baronaigien ja Ruskey [1988] (t-puu, solmujen painoihin perustuva koodaus)
- Skarbek [1988] (yleinen puu, ei koodausta)
- Er [1989] (rotaatioihin perustuva koodaus)
- Ruskey ja Proskurowski [1990] (tasapainotetut sulkujonot, Gray-koodi)
- Gupta [1991] (P-jonot)
- Roelants van Baronaigien [1991] (rotaatioihin perustuva koodaus)
- Lucas *et al.* [1993] (ei koodausta)
- Korsh [1994] (t-puu, Roelants van Baronaigienin [1991] menetelmän yleistys)
- Bapiraju ja Bapeswara Rao [1994] (Korshin [1993] koodausmenetelmä)
- Xiang *et al.* [1997] (kielioppeihin perustuva koodaus)
- Walsh [1998] (tasapainotetut sulkujonot)
- Korsh ja Lipschutz [1998] (t-puut, ‘siirtoihin’ (shift) perustuva algoritmi)
- Vajnovszki [1998] (solmujen painoihin perustuva koodaus).

Erilaisia puiden koodausmenetelmiä ja järjestyksiä soveltaen numerointi- ja vastaavia käänteisalgoritmeja ovat esittäneet mm. seuraavat:

- Knott [1977] (puupermutaatiot)
- Ruskey ja Hu [1977] (tasojonot)
- Rotem ja Varol [1978] (arpajonot (ballot sequences))
- Ruskey [1978] (t-puu, Ruskeyn ja Hun [1977] yleistys)
- Trojanowski [1978] (t-puu, puupermutaatioihin perustuva koodaus)
- Zaks ja Richards [1979] (t-puun muunnelma)
- Solomon ja Finkel [1980] (ei koodausta)
- Zaks [1980] (Zaksin jonot)

- Er [1985] (tasapainotetut sulkujonot)
- Pallo ja Racca [1985] (P-jonot; L-jonot)
- Proskurowski ja Ruskey [1985] (tasapainotetut sulkujonot, Gray-koodi)
- Pallo [1986] (painojonot)
- Roelants van Baronaigien ja Ruskey [1988] (t-puu, solmujen painoihin perustuva koodaus)
- Lucas *et al.* [1993] (ei koodausta).

5. Satunnaisten binääripuiden generointi

Tässä luvussa kuvataan yksityiskohtaisesti satunnaisia binääripuita generoivat algoritmit sekä eräitä yleisempiä puita generoivia algoritmeja. Luvun lopussa on satunnaisten puiden generointialgoritmien sekä muutoin aihetta läheisesti sivuavien julkaisujen kirjallisuusluettelo.

5.1. Menetelmien luokittelu

Satunnaisten binääripuiden generointi on erikoistapaus kombinatoristen rakenteiden satunnaisesta generoinnista, jonka uranuurtajia olivat Nijenhuis ja Wilf [1975]. Kombinatoristen rakenteiden satunnainen generointi liittyy läheisesti eräisiin muihin klassisiin kombinatoriikan perusprobleemoihin, jotka mainittiin johdantoluvussa. Furnas [1984] listasi kombinatoristen rakenteiden satunnaisen generoinnin viisi perusstrategiaa. Seuraavassa esitämme tämän jaottelun yksinkertaistettuna ja soveltaen sitä binääripuiden tapaukseen. Merkitsemme n:llä puun solmujen lukumäärää.

1) Satunnaisesti päättyvä luettelointi. Jos määrätyn koodausmenetelmän puitteissa käytettävissä on ainoastaan binääripuiden luettelointialgoritmi, voimme valita satunnaisen kokonaisluvun $k \in [1,C_n]$ ja antaa luettelointialgoritmin tehdä k seuraavan puun generointioperaatiota. Tämän menetelmän keskimääräinen aikavaatimus on eksponentiaalinen.

2) Käänteinen numerointioperaatio. Arvotaan satunnainen järjestysnumero väliltä $[1,C_n]$ ja konstruoidaan järjestysnumeroa vastaava binääripuu. Menetelmä käyttää eksponentiaalisia lukuja.

3) Ositusmenetelmä/hajoita ja hallitse -menetelmä. Tapausavaruus jaetaan osiin, ja osille lasketaan oikeat todennäköisyydet. Valittuun osaan sovelletaan rekursiivisesti ositusmenetelmää. Esimerkki tällaisesta menetelmästä esitetään kohdassa 5.2, jossa binääripuiden joukko jaetaan osiin sen perusteella, montako solmua vasemmassa ja oikeassa alipuussa on.

4) Satunnaiset lisäysoperaatiot. Periaate on seuraava: oletetaan satunnainen k-solmuinen binääripuu. Satunnainen lisäysoperaatio lisää yhden solmun puuhun siten, että kaikki k+1 -solmuiset puut ovat yhtä todennäköisiä. Solmuja lisätään puuhun, kunnes siinä on n solmua. Esimerkki tällaisesta on Rémyn algoritmi (kohta 5.3).

5) Bijektiiviset menetelmät. Monissa tapauksissa on mahdollista löytää 1-1

-vastaavuus (‘koodaus’) kombinatorisen rakenteen ja jonkin helpommin hallittavan rakenteen välille. Koodauksia generoidaan satunnaisesti, ja muunnetaan ne lopulliseen muotoon. Binääripuiden koodauksia käsiteltiin omassa luvussaan, ja tässä luvussa esitetään useita algoritmeja, jotka generoivat satunnaisia binääripuita hyödyntäen jotakin koodausmenetelmää.

Kategorian 5 ‘normaalimenettelynä’ pidämme sellaista, jossa binääripuuta vastaavan koodisanan merkkejä arvotaan yksi kerrallaan ja ennen kunkin merkin valintaa lasketaan eri vaihtoehtojen todennäköisyydet sitä silmälläpitäen, että kaikki mahdolliset lopputuloksena saatavat koodisanat ovat yhtä todennäköisiä. Tällaisia ovat Arnoldin ja Sleepin [1980], Martinin ja Orrin [1990] sekä Johnsenin [1991] algoritmit. Koodausta käyttävä algoritmi voi samalla kuulua myös johonkin muuhun esitetyistä kategorioista. Atkinsonin ja Sackin [1992] algoritmi kuuluu samalla myös kategoriaan 3. Korshin [1993] ja Atkinsonin [1993] algoritmeja voitaisiin kutsua myös kutsua ‘rotaatiopohjaisiksi’.

Alonson ja Schottin [1995] esittämä jaottelu on olennaisesti sama kuin yllä kuitenkin sillä erotuksella, että he eivät tarkastele kohtia 1) ja 2), ja omana kategorianaan mainitsevat ns. *hylkäysmenetelmän* (method by rejection): joillekin puurakenteille tehokkain tunnettu algoritmi generoi muitakin rakenteita kuin niitä, joita halutaan; mikäli rakenne ei ole halutun tyyppinen, se hylätään ja generoidaan uusi [Alonso and Schott, 1995, Ch. 7; Barcucci, 1994].

Yleisen menetelmän kombinatoristen rakenteiden satunnaista generointia varten esittivät Flajolet *et al.* [1993; 1994]. Laskennallisen vaativuuden näkökulmasta kombinatoristen rakenteiden satunnaista generointia ovat tutkineet Jerrum *et al.* [1986]. Hickey ja Cohen [1983] esittivät yleisen menetelmän satunnaisen kontekstittomaan kielioppiin kuuluvan sanan generoimiseksi: eräät kombinatoriset rakenteet (esim. binääripuu) voidaan esittää kieliopin avulla sopivaa koodausta käyttäen. Mairson [1994] paransi Hickeyn ja Cohenin algoritmin saavuttamaa aika- ja tilavaatimusta. Alonso *et al.* [1997] (ks. myös Alonso ja Schott [1995]) esittivät ‘malleihin’ (pattern) perustuvan yleisen menetelmän erilaisten satunnaisten puiden generoimiseksi. Edellämainittuja ‘puhtaasti’ yleisiä menetelmiä ei tässä tutkielmassa yksityiskohtaisesti tarkastella.

Jokaista binääripuiden käänteistä numerointialgoritmia voidaan periaatteessa käyttää myös binääripuiden satunnaiseen generointiin. Arvotaan järjestysnumero väliltä $1...C_n$, missä n on solmujen lukumäärä, ja konstruoidaan järjestysnumeroa vastaava

puu. Algoritmin sisältämien operaatioiden ja laskutoimitusten kannalta ongelmallista on kuitenkin se, että C_n kasvaa n:n suhteen eksponentiaalisesti; esimerkiksi luvun C_{5000} desimaaliesitys sisältää yli 2000 numeroa [Martin and Orr, 1990]. Tämä ongelma ei ole pelkästään teoreettinen, sillä eräät keskeiset puihin liittyvät ominaisuudet ovat sellaisia, että niiden arvot kasvavat hitaasti puun solmujen lukumäärän funktiona (esim. puun korkeus), joten testitilanteessa niitä koskevan asymptoottisen informaation selville saaminen saattaa edellyttää suuria solmulukumääriä.

5.2. Hajoita ja hallitse -algoritmi

Seuraavassa esitetään yksinkertainen satunnaisia binääripuita generoiva algoritmi, jossa solmut lisätään puuhun esijärjestyksessä, ja jokaisen solmun kohdalla arvotaan, montako solmua tulee solmun vasempaan alipuuhun ja montako oikeaan (algoritmi 5.1). Arvonnassa käytettävät todennäköisyydet lasketaan niin, että kaikki erilaiset lopputuloksena saatavat puut ovat yhtä todennäköisiä. Todennäköisyyksien laskenta tapahtuu Catalanin lukujen avulla, joten laskennassa käytettävät luvut ovat solmujen lukumäärän suhteen eksponentiaalisia. Algoritmi ei käytä mitään binääripuiden koodausmenetelmää. Kutsu on muotoa RandomBinaryTree(n,Root), missä n on puun solmujen lukumäärä ja Root puun juurisolmu. Algoritmi on johdettavissa suoraan tunnetusta binääripuiden lukumäärän kaavasta 5.2, joten algoritmin ensimmäinen keksijä ei ole tiedossa. Algoritmi esiintyy muun muassa Furnasin [1984] artikkelissa.

```
procedure RandomBinaryTree(n, Solmu)
valitaan satunnainen k∈[0,n-1], missä kunkin k:n todennäköisyys on C_k C_{n-1-k}/C_n
if (k ≠ 0)      then   luodaan Solmun vasen lapsi;
                       RandomBinaryTree(k, Solmun vasen lapsi)      end if
if (n-1-k ≠ 0) then    luodaan Solmun oikea lapsi;
                       RandomBinaryTree(n-1-k, Solmun oikea lapsi)  end if
end procedure
```

Algoritmi 5.1. Satunnaisia binääripuita generoiva algoritmi.

Jos mielivaltaisessa puussa tai alipuussa on n solmua ja sen juuren vasemmassa alipuussa on $k \in [0,n-1]$ solmua, niin juuren oikeassa alipuussa on n-1-k solmua. Erilaisten vasempien alipuiden lukumäärä on C_k ja oikeiden C_{n-1-k}, joten erilaisia

mahdollisuuksia on C_kC_{n-1-k}. Kun tämä tarkastelu tehdään kaikilla mahdollisilla k:n arvoilla, saadaan tulokseksi n-solmuisten binääripuiden lukumäärä [esim. Knuth, 1973a, s. 388]:

(5.2) $$\sum_{k=0}^{n-1} C_kC_{n-1-k} = C_n.$$

Koska kaikkien n-solmuisten puiden todennäköisyys on sama, kunkin k:n todennäköisyys on C_kC_{n-1-k}/C_n. Seuraavassa esitetään esimerkit sovellettavista todennäköisyyksistä.

n	0	1	2	3	4	5	6	7	8	9	10
C(n)	1	1	2	5	14	42	132	429	1430	4816	16798

Taulukko 5.3. Catalanin luvun arvoja.

	k=0	1	2	3	4	5	6
n=1	1						
2	1/2	1/2					
3	2/5	1/5	2/5				
4	5/14	2/14	2/14	5/14			
5	14/42	5/42	4/42	5/42	14/42		
6	42/132	14/132	10/132	10/132	14/132	42/132	
7	132/429	42/429	28/429	25/429	28/429	42/429	132/429

Taulukko 5.4. Tiheysfunktion $p(n,k)=C_kC_{n-1-k}/C_n$ arvoja.

Taulukossa 5.3 ovat Catalanin luvut C_0-C_{10}. Taulukon 5.4 alkiot p(n,k) kuvaavat eri k:n arvojen todennäköisyydet. Taulukon 5.4 kukin rivi on tiheysfunktio k:n suhteen, ts. jokaisen rivin summa on 1. Taulukossa 5.5 ovat vastaavat kertymäfunktion P(n,k) arvot.

	k=0	1	2	3	4	5	6
n=1	1						
2	1/2	2/2					
3	2/5	3/5	5/5				
4	5/14	7/14	9/14	14/14			
5	14/42	19/42	23/42	28/42	42/42		
6	42/132	56/132	66/132	76/132	90/132	132/132	
7	132/429	174/429	202/429	227/429	255/429	297/429	429/429

Taulukko 5.5. Kertymäfunktion P(n,k) arvoja.

Satunnainen k:n arvo voidaan nyt tuottaa satunnaisluvun $x \in [0,1)$ avulla: etsitään riviltä n kertymäfunktion P(n,k) pienin arvo, joka ylittää satunnaisluvun. Suurilla n:n arvoilla taulukkojen käyttäminen on luonnollisestikin mahdotonta, sillä niiden tilavaatimus on $O(n^2)$, samoin kuin niiden rakentamiseen tarvittava aika.

Kertymäfunktion P(n,k) ratkaistussa muodossa esitti Sprugnoli [1992]:

$$Q(n,k) = \frac{1}{2} - \frac{n-2k}{2n}\binom{2k}{k}\binom{2(n-k)}{n-k}\binom{2n}{n}^{-1}, \tag{5.6}$$

missä P(n,k)=Q(n,k+1) eli funktion Q toisen parametrin mahdolliset arvot voivat olla 1...n. Myös funktio Q käyttää eksponentiaalisia lukuja, ja tämän ongelman kiertämiseksi Sprugnoli saavutti numeerisia menetelmiä (Newton-Raphson -method) soveltamalla laskukaavat, jotka esitetään seuraavassa sellaisenaan. Olkoon N suurin mahdollinen kokonaisluku, johon asti voidaan taulukoida lausekkeen $\binom{2n}{n}$ arvot: käytännössä taulukon koko on muutamia kymmeniä. Nyt Q(n,k) evaluoidaan seuraavasti:

1) Jos n≤N, niin käytämme kaavaa 5.6.

2) Jos n>N ja r=min(k,n-k)≤N, niin w=max(k,n-k) ja lausekkeen taulukoimattomat osat saadaan kaavasta

$$\binom{2w}{w}\binom{2n}{n}^{-1} = \sqrt{n/w}\,4^{-r}\,\alpha(w)/\alpha(n),$$

missä korjaussarjan $\alpha(x)=1-1/(8x)+1/(128x^2)+5/(1024x^3)-...$ termejä lasketaan riittävän pitkälle halutun tarkkuuden saavuttamiseksi.

3) Jos n, k ja n-k ovat suurempia kuin N, niin

$$Q(n,k) = \frac{1}{2} - \frac{2n-k}{2\sqrt{nk(n-k)p}}\,\frac{a(k)a(n-k)}{a(n)}.$$

Aikavaatimus funktion arvon laskemiselle annettuun tarkkuuteen asti kohdissa 1)-3) on O(1) [Sprugnoli, 1992].

Algoritmi 5.1 on havainnollinen ja yksinkertainen ja se on hyvä johdatus binääripuiden lukumäärän olemukseen. Samalla se kuitenkin on havainnollistus eksponentiaalisten lukujen ongelmasta, johon ‘suoraviivaisella’ lähestymistavalla helposti törmätään kombinatorisia rakenteita käsittelevien algoritmien yhteydessä. Satunnaisten binääripuiden generoinnin kannalta (pieniä n:n arvoja lukuunottamatta) algoritmi 5.1 sekä laskentakaavat 1-3 ovat hankalia verrattuna jäljempänä esitettäviin binääripuiden koodauksia hyödyntäviin generointialgoritmeihin.

5.3. Satunnaiset solmujen lisäykset (Rémy)

Rémy esitti vuonna 1985 algoritmin, joka muodostaa satunnaisen lehtisolmut sisältävän binääripuun lisäämällä solmuja satunnaiseen kohtaan puussa. Algoritmi ei eksplisiittisesti käytä mitään binääripuiden koodausmenetelmää, joskin algoritmia toteutettaessa solmujen ‘saavuttamisen’ ongelma johtaa esim. solmujen väliaikaiseen tallettamiseen taulukoihin. Seuraavassa kuvataan algoritmi ja todistetaan se oikeaksi. Tämä luku perustuu pääosin Alonson ja Schottin [1995] teokseen.

Oletetaan binääripuu, jossa on $k \geq 0$ sisäsolmua ja $k+1$ lehteä. Valitaan satunnaisesti yksi puun $2k+1$ solmusta; olkoon se solmu v. ‘Irrotetaan’ väliaikaisesti v mahdollisine alipuineen puusta. Lisätään v:n paikalle uusi sisäsolmu, jonka joko oikeaksi tai vasemmaksi lapseksi lisätään uusi lehtisolmu. Valinta oikean ja vasemman välillä tehdään satunnaisesti. Uuden solmun toiseksi lapseksi liitetään solmu v mahdollisine alipuineen. Näitä sisäsolmu-lehtisolmuparien lisäyksiä toistetaan kunnes puussa on n sisäsolmua.

Algoritmin oikeaksi todistamisessa käytetään apuna lehtisolmujen numerointia. Ensimmäisenä lisättävän lehtisolmun numero on 1. Uutta lehtisolmua lisättäessä numeroidaan se yhtä suuremmalla numerolla kuin edellisen lisätyn lehtisolmun numero. Koska n-solmuisen binääripuun lehtisolmut voidaan numeroida $(n+1)!$ eri tavalla, erilaisten n-solmuisten puiden määrä on $C_n(n+1)!=(2n)!/n!$.

Lause 5.7. [Alonso and Schott, 1995] Rémyn algoritmi generoi jokaisen erilaisen numeroidut lehtisolmut sisältävän binääripuun todennäköisyydellä $n!/(2n)!$.

Todistus. Sovelletaan täydellistä induktiota. Alkutilanteessa $k=0$ binääripuu on ykkösellä numeroitu lehti. Algoritmi generoi molemmat puut, joissa on 1 sisäsolmu ja 2 lehtisolmua, todennäköisyydellä $1!/2!=1/2$. Oletetaan, että algoritmi generoi kaikki puut, joissa on k sisäsolmua, todennäköisyydellä $k!/(2k)!$. Olkoon T_{k+1} mielivaltainen binääripuu, jossa on $k+1$ sisäsolmua. Etsitään siitä suurimman numeron $k+2$ omaava lehti u. Olkoon u:n vanhempisolmu w ja w:n toinen lapsi v. Muodostetaan puu T_k poistamalla puusta T_{k+1} solmut u ja w ja liittämällä solmu v mahdollisine alipuineen solmun w paikalle. Ainoa tapa generoida puu T_{k+1} on ensin konstruoida puu T_k (jonka todennäköisyys induktio-oletuksen nojalla on $k!/(2k)!$), valita solmu v (todennäköisyys $1/(2k+1)$) ja valita oikea suunta (todennäköisyys 1/2). Puun T_{k+1} todennäköisyys on edellämainittujen todennäköisyyksien tulo $(k+1)!/(2k+2)!$, joten lause seuraa. ?

Algoritmin tehokkuuden kannalta ratkaisevaa on se, kuinka toteutetaan puun satunnaisesti valitun solmun v etsiminen, koska uuden sisä- ja lehtisolmun lisääminen puuhun voidaan tehdä vakioajassa. Alonson ja Schottin [1995] esimerkkitoteutuksessa turvaudutaan suorasaantirakenteeseen: puu rakennetaan 2n+1 -kokoiseen taulukkoon, jonka alkiot sisältävät puun solmut, särmät ja numeroinnin, jotka esitetään kokonaislukuina. Mikäli satunnaisen solmun etsimiskustannuksen taulukosta katsotaan olevan O(1), algoritmin aikavaatimus on O(n).

Seuraavassa esitämme arvion aikavaatimukselle myös siinä tapauksessa, että puun solmujen tallentamiseen ei käytetä suorasaantirakennetta. Oletetaan hypoteettinen, riittävän kevyt metodi, jonka avulla osaamme virheettömästi suunnistaa puun juuresta särmiä pitkin lisäyskohtaan. Kun kaikki n-solmuiset binääripuut ovat yhtä todennäköisiä, keskimääräinen kuljettu polunpituus juuresta satunnaisesti valittuun solmuun ja näinollen myös aikavaatimus ovat $O(\sqrt{n})$ [esim. Gonnet ja Baeza-Yates, 1991, s. 96]. Lisäksi yleisesti pätee $1^m+2^m+3^m+...+n^m=O(n^{m+1})$, koska $\int_1^n x^m dx = O(n^{m+1})$: diskreetin funktion arvojen summaa voidaan arvioida vastaavan jatkuvan funktion määrätyn integraalin ja pinta-alan välisen yhteyden kautta [esim. Knuth, 1973a; Graham *et al*., 1989]. Aikavaatimus on siis keskimäärin $O(n^{1.5})$.

5.4. Tasapainotetut sulkujonot (Arnold ja Sleep)

Arnold ja Sleep [1980] esittivät ensimmäisen satunnaisia binääripuita generoivan algoritmin, joka ei ole eksponentiaalinen minkään resurssin suhteen. Algoritmi tuottaa tasapainotettuja sulkujonoja, joiden pituus on 2n. Toimintaperiaate on seuraava. Alku- ja loppusulkuja arvotaan järjestyksessä vasemmalta oikealle. Arvonnassa käytettävät todennäköisyydet lasketaan generoinnin edetessä erikseen jokaisen merkin kohdalla siten, että kaikki mahdolliset lopputuloksena saatavat tasapainotetut sulkujonot ovat yhtä todennäköisiä. Ainoan ongelman siis muodostaa generointiprosessin aikana tapahtuva todennäköisyyksien selvittäminen, ja siihen perehdymme seuraavassa.

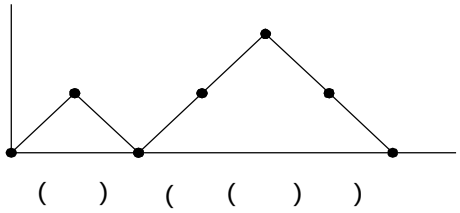
Tasapainotetun sulkujonon jokainen alkuosa (prefix) sisältää vähintään yhtä monta alkusulkua kuin loppusulkua. Oletetaan, että olemme konstruoimassa tasapainotettua sulkujonoa, ja 1) jonon valmiina olevassa alkuosassa on r avointa alkusulkua eli sellaista alkusulkua, joita vastaava loppusulku toistaiseksi puuttuu, ja 2) jonon generoimatta olevan loppuosan pituus on k. Erilaisten mahdollisten loppuosien

lukumäärä riippuu vain r:stä ja k:sta, mutta ei alkuosan pituudesta. Merkitsemme tätä lukumäärää A(r,k):lla. Tällöin A(r+1,k-1) on niiden erilaisten mahdollisten loppuosien lukumäärä, jotka alkavat alkusululla ja A(r-1,k-1) niiden erilaisten mahdollisten loppuosien lukumäärä, jotka alkavat loppusululla. Koska kaikkien erilaisten mahdollisten lopputuloksena saatavien jonojen on oltava yhtä todennäköisiä, A:n avulla ilmaistu todennäköisyys P(r,k) sille, että seuraavaksi tulostetaan loppusulku, on

$$\mathrm{P(r,k)} = \frac{A(r-1,k-1)}{A(r,k)}. \tag{5.8}$$

Seuraavassa rakennetaan välineistöä, jonka avulla funktio A voidaan löytää. Vastatkoon alkusulku lukua 1 ja loppusulku lukua -1. Tarkastellaan mielivaltaista valmista sulkujonon loppuosaa ja indeksoidaan sen alkiot numeroilla 1...k. Muodostetaan vastaava (k+1)-alkioinen kumulatiivinen *summajono* S seuraavasti: S(0)=r; S(i)=S(i-1)+x, missä i läpikäy arvot 1...k ja x on kulloinkin 1 tai -1 sen mukaan, mitä sulkujonossa on indeksin i kohdalla. Esimerkiksi tasapainotettua sulkujonoa ()(()) vastaava summajono on (0,1,0,1,2,1,0), ja alkutilanteessa r=0 ja k=6. Loppuosaa ()) vastaava summajono on (1,2,1,0), ja alkutilanteessa r=1 ja k=3. Jono)(() ei ole tasapainotettu, kun alkutilanteessa r=0; summajono on (0,-1,0,1,0).

Summajonon geometrinen havainnollistus tapahtuu koordinaatistoon piirretyn jatkuvan murtoviivan avulla seuraavasti: merkitään koordinaatiston pisteet (i,S(i)), i=0...k, ja yhdistetään kukin peräkkäinen piste särmällä (kuva 8). Polun pisteiksi kutsumme vain näitä erikseen merkittyjä pisteitä. Tuloksena on siis yhtenäinen *polku* (path, zigzag line), missä jokainen kahta peräkkäistä pistettä yhdistävä särmä on saman pituinen ja suuntautuu joko oikealle ylös (vastaten alkusulkua) tai oikealle alas (vastaten loppusulkua).



Kuva 8. Tasapainotettua sulkujonoa ()(()) vastaava polku.

Polun alkupiste R=(0,r) määräytyy avointen alkusulkujen lukumäärän r perusteella. Tasapainotetun sulkujonon ominaisuuksista seuraa, että polku päättyy pisteeseen K=(k,0), eikä sen yksikään piste sijaitse y-akselin negatiivisella puolella. Kutsumme sellaista *positiiviseksi poluksi*, ja funktion A(r,k) arvo on kaikkien erilaisten

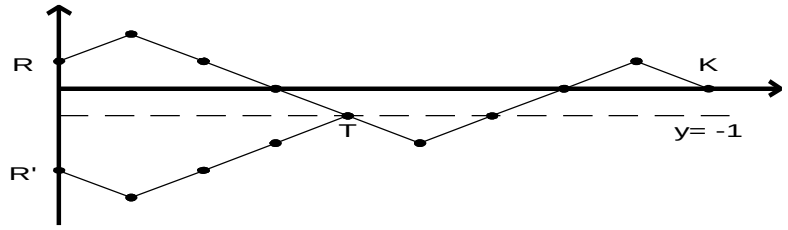
positiivisten polkujen lukumäärä. *Negatiivisen* polun alku- ja loppupisteet ovat samat, mutta sen yksi tai useampi piste sijaitsee y-akselin negatiivisella puolella. Merkitään U(r,k):lla kaikkien erilaisten pisteitä R ja K yhdistävien negatiivisten polkujen lukumäärää. Nyt T(r,k)=A(r,k)+U(r,k) on kaikkien erilaisten em. pisteitä yhdistävien polkujen lukumäärä, eli

(5.9) $$A(r,k) = T(r,k)-U(r,k).$$

Jokainen pisteitä R ja K yhdistävä polku sisältää täsmälleen p alas suuntautuvaa särmää ja täsmälleen q ylös suuntautuvaa särmää siten, että p+q=k ja p-q=r (tästä voimme laskea p=(k+r)/2). T(r,k) on toisaalta vastaus kysymykseen ’monellako tavalla p loppusulkua voidaan sijoitella p+q positioon, kun jonon ei tarvitse olla tasapainotettu?’, eli

(5.10) $$T(r,k) = \binom{p+q}{p} = \binom{k}{(k+r)/2}.$$

Seuraavaksi johdamme U(r,k):n ilmaistuna k:n ja r:n avulla. Tarkastellaan mielivaltaista pisteitä R ja K yhdistävää negatiivista polkua (kuva 9). Etenemme polkua pisteestä R, kunnes olemme ensimmäisen kerran alittaneet y-akselin ja saapuneet pisteeseen T=(t,-1). Jokaisella negatiivisella polulla tällaisia pisteitä on täsmälleen yksi. Seuraavaksi piirrämme peilikuvan juuri kulkemastamme pisteiden R ja T välisestä osapolusta suoran y=-1 suhteen: sen alkupiste on R’=(0,-r-2) ja loppupiste T. Näin menetellen jokaista negatiivista polkua RTK vastaa yksi ja vain yksi polku R’TK, missä osapolut RT ja R’T ovat toistensa peilikuvia ja osapolut TK ovat molemmissa samat. Toisin sanoen, polkujen RTK ja R’TK välillä vallitsee 1-1 -vastaavuus.



Kuva 9. Negatiivinen polku RTK ja sen vastine R’TK.

Erilaisten polkujen R’TK lukumäärä pystytään laskemaan samalla tavalla kuin T(r,k) laskettiin. Käännetään kuvan 9 merkinnät ympäri suoran y=0 suhteen, jolloin polun R’TK alkupiste on (0,r+2) ja loppupiste sama kuin edellä. Polku voi olla positiivinen tai negatiivinen. Nyt erilaisten polkujen R’TK lukumäärä (ja 1-1

-vastaavuuden nojalla myös U(r,k)) saadaan suoraan kaavasta 5.10:

(5.11) $$\mathrm{U(r,k)} = \binom{k}{(k+r+2)/2}.$$

Kaavojen 5.9, 5.10 ja 5.11 perusteella lausekkeen T(r,k)-U(r,k) arvoksi [Arnold and Sleep, 1980] saadaan sievennettynä

(5.12) $$\mathrm{A(r,k)} = \frac{2(r+1)}{(k+r+2)}\binom{k}{(k+r)/2},$$

ja lopputulokseksi kaavojen 5.8 ja 5.12 perusteella

(5.13) $$\mathrm{P(r,k)} = \frac{r(k+r+2)}{2k(r+1)}.$$

Satunnainen tasapainotettu sulkujono voidaan nyt rakentaa vasemmalta oikealle seuraavasti: jokaisen generoitavan merkin kohdalla arvotaan satunnaisluku väliltä [0,1] ja lasketaan P(r,k) kulloisillakin r:n ja k:n arvoilla. Jos satunnaisluku on pienempi kuin P(r,k), tulostetaan loppusulku, muussa tapauksessa alkusulku.

Koska n-solmuista binääripuuta vastaa 2n-merkkinen sulkujono ja kaavan 5.13 muuttujien r ja k suuruus on O(n), Arnoldin ja Sleepin algoritmi tuottaa tasapainotettuja sulkujonoja ajassa O(n) ja algoritmissa käytettyjen lukujen suuruus on enintään $O(n^2)$ [Mäkinen, 1999].

5.5. Tasapainotetut sulkujonot (Atkinson ja Sack)

Atkinson ja Sack [1992] esittivät satunnaisia tasapainotettuja sulkujonoja generoivan algoritmin, joka toimii lineaarisessa ajassa eikä käytä suurempia lukuja kuin 2n, missä n on vastaavan binääripuun solmujen lukumäärä. Algoritmin idea on lähteä sulkujonosta, jonka ei tarvitse olla tasapainotettu, ja muuntaa se tasapainotetuksi sulkujonoksi rekursiivisella hajoita ja hallitse -tyyppisellä operaatiolla. Seuraavassa käsitellään Atkinsonin ja Sackin algoritmiin liittyviä määritelmiä, käsitteitä ja todistuksia, ja kuvataan itse algoritmi.

Olkoon B_n joukko, joka sisältää ne sulkujonot, joiden pituus on 2n ja jotka koostuvat n alkusulusta ja n loppusulusta. Rajoituksia sulkujen järjestyksen suhteen ei ole. Käytämme käsitteitä kumulatiivinen summajono ja polku, jotka on esitelty Arnoldin ja Sleepin [1980] algoritmia käsittelevässä kohdassa. Jokaista joukon B_n sulkujonoa vastaava summajono alkaa nollalla ja loppuu nollaan, ja polku alkaa origosta ja päättyy pisteeseen (2n,0). Olkoon tällaisen sulkujonon *virhe* (defect)

vastaavan summajonon negatiivisten alkioiden lukumäärä. Jos sulkujonon virhe on i, summajonossa on i negatiivista alkiota ja polku sisältää 2i y-akselin alapuolella sijaitsevaa särmää. Tasapainotettua eli ‘virheetöntä’ sulkujonoa vastaava summajono ei sisällä negatiivisia lukuja: sen virhe on nolla.

Sisältäköön joukon B_n osajoukko B_{ni} ne sulkujonot, joiden virhe on i. Nyt joukot B_{n0}, B_{n1}, ... , B_{nn} ovat erillisiä ja niiden unioni on B_n. Joukko B_{n0} vastaa tasapainotettuja sulkujonoja, joten sen alkioiden lukumäärä on C_n. Kaikki joukot B_{ni} ovat saman kokoisia [Feller, 1957, s. 72; Atkinson and Sack, 1992]. Tähän perustuen on mahdollista luoda 1-1 -vastaavuus joukon B_{n0} alkioiden ja minkä tahansa joukon B_{ni} alkioiden välille. Satunnainen tasapainotettu sulkujono voidaan nyt tuottaa valitsemalla satunnainen joukon B_n alkio ja etsimällä sitä vastaava joukon B_{n0} alkio.

Oletetaan mielivaltainen sulkujono $w \in B_n$. Tällöin käytämme merkintää w* sulkujonosta, joka on saatu w:stä korvaamalla jokainen alkusulku loppusululla ja loppusulku alkusululla. Merkkijono w voidaan jakaa ‘tekijöihin’ w_1 ja w_2, jos $w=w_1w_2$, missä w_1 sisältää yhtä monta alkusulkua kuin loppusulkua ja w_2 sisältää yhtä monta alkusulkua kuin loppusulkua eikä kumpikaan ole tyhjä. Jos tällaista osiinjakoa ei voida tehdä, w on *jakamaton* (irreducible).

Lemma 5.14. [Atkinson ja Sack, 1992] Jokainen jono w voidaan jakaa tekijöihin $w_1w_2...w_k$ yhdellä ja vain yhdellä tavalla, missä jokainen w_i on jakamaton.

Ylläoleva ominaisuus voidaan helposti todeta esim. hakemalla sulkujonoa vastaavan polun kohdat, joissa y=0. Jos w_i on jakamaton, sitä kuvaava polku pysyy alku- ja loppupisteitä lukuunottamatta koko ajan y-akselin positiivisella tai negatiivisella puolella. Silloin kun polku kulkee y-akselin positiivisella (vast. negatiivisella) puolella, niin w_i (vast. w_i*) on tasapainotettu.

Seuraavassa kuvaamme Atkinsonin ja Sackin algoritmin esiteltyihin merkintöihin ja määritelmiin perustuen. Ensin generoidaan satunnainen sulkujono $w \in B_n$ esimerkiksi poimimalla indeksijoukosta {1,2,...,2n} satunnaisesti n alkiota [esim. Reingold *et al.*, 1977] ja merkitsemällä näiden indeksien osoittamiin sulkujonon alkioihin alkusulku ja muihin loppusulku. Tämä on selvästikin mahdollista lineaarisessa ajassa.

Algoritmin ydin on kuvaus joukolta B_n joukolle B_{n0} (algoritmi 5.15). Parametrina annetun sulkujonon $w \in B_n$ perusteella funktio palauttaa arvonaan samanpituisen

sulkujonon, joka on w:n kuva joukossa B_{n0} eli tasapainotettu sulkujono. Merkki λ tarkoittaa tyhjää merkkijonoa, ja selvyyden vuoksi sulkujonon alkiot ilmaistaan aaltosulkeilla { } (rivit 8 ja 9).

```
(1)   function P(w)
(2)   comment Atkinson ja Sack [1992]: sulkujonon w∈Bn kuvaus joukolle Bn0
(3)   if (w = λ)
(4)   then   P := λ
(5)   else   muodosta osajonot u ja v siten, että uv=w ja u≠λ on jakamaton
(6)          if (u on tasapainotettu)
(7)          then   P := u P(v)
(8)          else   comment nyt on u = }t{
(9)                 P := { P(v) } t*
(10)         end if
(11)  end if
(12)  return P
(13)  end function
```

Algoritmi 5.15. Tasapainotetun sulkujonon generointi sulkujonon $w \in B_n$ perusteella.

Rivillä 5 sulkujonon w osajono u löydetään esim. hakemalla jonoa w vastaavan summajonon kaksi vasemmanpuoleisinta nolla-alkiota. Rivillä 6 jakamattoman sulkujonon ensimmäinen merkki ilmaisee, onko jono tasapainotettu.

Todistetaan seuraavassa, että P on bijektio joukolta B_{ni} joukolle B_{n0}. Merkinnällä $\|x\|$ tarkoitamme merkkijonon x pituutta jaettuna kahdella.

Lause 5.16. [Atkinson ja Sack, 1992] Olkoon $w_1, w_2 \in B_{ni}$. Jos $P(w_1)=P(w_2)$, niin $w_1=w_2$ eli P on injektio.

Todistus. Oletetaan, että $P(w_1)=P(w_2)$. Olkoon $w_k=u_kv_k$, missä u_k on jakamaton ja ei-tyhjä, k=1,2. Muuta ei oleteta pituuksista $\|u_1\|$ ja $\|u_2\|$. On neljä mahdollisuutta:

(1) Kukin u_k on tasapainotettu. Tällöin on $P(w_k)=u_kP(v_k)$, missä v_k:n virhe on i. Oletuksen nojalla on $u_1P(v_1)=u_2P(v_2)$. Nyt Lemman 5.14 nojalla seuraa $u_1=u_2$ ja edelleen $P(v_1)=P(v_2)$ ja $\|v_1\|=\|v_2\|=s$. Mutta nyt $v_1, v_2 \in B_{si}$ ja $P(v_1)=P(v_2)$, joten rekursiivisesti Lauseen 5.16 nojalla $v_1=v_2$. Oletuksesta seuraa siis $w_1=w_2$.

(2) Mikään u_k=}t_k{ ei ole tasapainotettu. Tällöin on $P(w_k)$={$P(v_k)$}t_k*, missä v_k:n virhe on i-||u_k||. Oletuksen nojalla on {$P(v_1)$}t_1*={$P(v_2)$}t_2*. Jonon {$P(v_k)$} jakamattomuus seuraa siitä, että tasapainotetun sulkujonon $P(v_k)$ ympärillä on sulkupari, joten Lemman 5.14 nojalla $P(v_1)=P(v_2)$ ja t_1*=t_2*. Tästä seuraa $||v_1||=||v_2||=s$ ja $u_1=u_2$. Täten v_1:n ja v_2:n virhe on i-||u_1||=j. Mutta nyt kuten kohdassa (1), pätee $v_1,v_2 \in B_{sj}$ ja $P(v_1)=P(v_2)$, joten $v_1=v_2$. Siis oletuksesta seuraa $w_1=w_2$.

(3) u_1 on tasapainotettu ja u_2=}t_2{ ei ole tasapainotettu. Tällöin v_1:n virhe on i ja v_2:n virhe on i-||u_2||. Oletuksen nojalla on $u_1P(v_1)$={$P(v_2)$}t_2*. Lemman 5.14 nojalla pätee u_1={$P(v_2)$} ja $P(v_1)=t_2$*, koska {$P(v_2)$} on jakamaton. Jälkimmäisen yhtäsuuruuden perusteella on $||v_1|| = ||u_2||-1$, missä vähennettävä ykkönen vastaa sulkuparin pituutta jaettuna kahdella. Mutta koska v_2:n virhe on i-||u_2||, täytyy olla $||u_2|| \leq i$. Vastaavasti v_1:n virheen perusteella seuraa $i \leq ||v_1||$. Pituusväittämät yhdistäen saadaan $i \leq ||v_1|| = ||u_2||-1 < ||u_2|| \leq i$, eli ristiriita.

(4) u_1 ei ole tasapainotettu ja u_2 on tasapainotettu. Tapaus on mahdoton samoista syistä kuin (3).

Kohtien (1), (2), (3) ja (4) perusteella funktio P on injektio. ?

Lauseen 5.16 perusteella kaikki lähtöjoukon B_{ni} alkiot kuvautuvat eri alkioille tulosjoukossa B_{n0}, kun i on kiinnitetty. Tämä yleistyy koskemaan kaikkia i:n arvoja 0...n. Vielä on varmistuttava siitä, että jokaista tulosjoukon B_{n0} alkiota vastaa lähtöjoukossa alkio. Tämä onnistuu osoittamalla kyseisten joukkojen yhtäsuuruus, mikä on aiemmin tehty Fellerin [1957, s. 72] teoksessa. Esitetään tässä kuitenkin Atkinsonin ja Sackin vaihtoehtoinen todistus. Seuraavassa käytämme joukon X alkioiden lukumäärästä merkintää card(X).

Lause 5.17. [Atkinson ja Sack, 1992] On voimassa $card(B_{ni}) = card(B_{n0})$, i=0...n.

Todistus. Lauseen 5.16 perusteella $card(B_{ni}) \leq card(B_{n0})$. Tiedämme, että $card(B_n) = \binom{2n}{n}$ ja $card(B_{n0}) = \frac{1}{n+1}\binom{2n}{n} = C_n$. Oletetaan, että ainakin yhdellä i:n arvolla $card(B_{ni}) < card(B_{n0})$. Tällöin on $\binom{2n}{n} = card(B_n) = \sum_{i=0}^{n} card(B_{ni}) < (n+1)card(B_{n0}) = \binom{2n}{n}$, eli ristiriita. ?

Lauseiden 5.16 ja 5.17 nojalla kuvaus P : $B_{ni} \rightarrow B_{n0}$ on bijektio, kun i on kiinnitetty ja välillä 0...n. Kuvaus P : $B_n \rightarrow B_{n0}$ on surjektio, ja jokaista tulosjoukon alkiota vastaa täsmälleen n+1 lähtöjoukon alkiota.

Algoritmin aikavaatimus on O(n), ja Atkinson ja Sack laskevat sen seuraavasti. Olkoon T(n) operaatioiden lukumäärä. Jakamattoman osajonon u löytäminen, kun w=uv, tapahtuu kumulatiivisella summausmenetelmällä ajassa O(r), missä r on u:n pituus. Koska sen jälkeen on laskettava aikavaatimus jonolle v, saadaan rekursioyhtälö T(n)=O(r)+T(n-r), jonka ratkaisu on O(n). Algoritmi ei käytä suurempia lukuja kuin 2n.

5.6. Oikean etäisyyden koodisanat (Martin ja Orr)

Martin and Orr [1990] esittivät satunnaisia binääripuita generoivan algoritmin, jossa käytetty koodausmenetelmä on seuraava. Binääripuun juuren numero on nolla. Jos solmun numero on i, niin sen oikean lapsen numero on i ja vasemman lapsen i+1. Koodisana $(x_0,x_1,...,x_{n-1})$ muodostetaan lukemalla puun solmujen numerot esijärjestyksessä. Näin menetellen koodisanan alkiot täyttävät ehdot $x_0=0$ ja $x_i \leq x_{i-1}+1$. Puun oikean polun kaikkiin solmuihin tulee siis nolla.

Kun n on puun solmujen lukumäärä, Martinin ja Orrin algoritmi arpoo järjestyksessä n koodisanan numeroa erityisen todennäköisyyskertymäfunktion avulla siten, että kaikki mahdolliset lopputuloksena saatavat koodisanat ovat yhtä todennäköisiä.

Martin ja Orr käyttävät koodisanasta nimitystä käännöstaulu (inversion table), joka on (puu- tai tavallisten) permutaatioiden eräs esitysmuoto [Knuth, 1973b]. Koska sovellettava solmujen numerointi ilmaisee samalla myös solmun etäisyyden puun oikeasta polusta, voidaan koodausmenetelmästä käyttää myös nimitystä oikean(puoleisen) etäisyyden menetelmä (vrt. vasemman etäisyyden menetelmä, kohta 3.4).

Oletetaan, että olemme konstruoimassa satunnaista oikean etäisyyden koodisanaa, ja sen valmiina oleva alkuosa on $x_0,...,x_j$, missä $x_j=i$. Todennäköisyyskertymäfunktio F(k) määrittelee todennäköisyyden sille, että $x_{j+1} \in \{0,...,k\}$, missä $k \leq i+1$. Nyt F(k)=a/b, missä a on kaikkien erilaisten mahdollisten alkuosien $x_0,...,x_j$ lukumäärä ja b on kaikkien erilaisten mahdollisten alkuosien $x_0,...,x_{j+1}$ lukumäärä, missä x_{j+1} voi olla mikä tahansa joukon {0,...,k} alkio. Martinin ja Orrin perustavaa laatua oleva tulos on todennäköisyyskaava

(5.18) $$F(n,i,j,k) = \frac{(k+1)(n-j+i+2)!(2n-2j+k)!}{(i+2)(n-j+k+1)!(2n-2j+i+1)!},$$

missä n on koodisanan pituus, i on viimeksi lasketun koodisanan alkion arvo, j on laskettavan koodisanan alkion indeksi ja k on yläraja laskettavan koodisanan alkion arvolle. Kun valitaan satunnaisluku $x \in [0,1)$, koodisanan seuraavan alkion arvo on suurin sellainen kokonaisluku m, että $x \geq F(n,i,j,m-1)$. Kaava 5.18 ei luonnollisestikaan sellaisenaan ole käytännössä käyttökelpoinen sisältämiensä kertomafunktioiden vuoksi.

Todennäköisyys, että k on seuraava koodisanan alkion arvo, on P(n,i,j,k) = F(n,i,j,k)-F(n,i,j,k-1). Määritellään lisäksi Q(n,j,k) = P(n,i,j,k-1)/P(n,i,j,k). Martin ja Orr johtavat kaavat

(5.19) $$Q(n,j,k) = \frac{(k+1)(n-j+k+1)}{(k+2)(2n-2j+k-1)},$$

ja

$$P(n,i,j,i+1) = \frac{(i+3)(n-j)}{(i+2)(2n-2j+i+1)}.$$

Huomaa, että kaavassa 5.19 ei ole i:tä, eikä kumpikaan kaava sisällä kertomafunktioita. Lisäksi pätee

$$P(n,i,j,k-1) = P(n,i,j,k)\, Q(n,j,k).$$

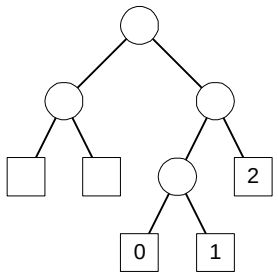
Nyt funktion P(n,i,j,k) arvot voidaan helposti laskea perättäisillä vähenevillä k:n arvoilla aloittaen suurimmasta mahdollisesta arvosta k=i+1.

Edelläkuvatulla tavalla on mahdollista generoida satunnaisia oikean etäisyyden sanoja ajassa O(n), missä n on vastaavan binääripuun solmujen lukumäärä [Martin and Orr, 1990], käyttäen lukuja, jotka ovat enintään $O(n^2)$ [Mäkinen, 1999].

5.7. Oksastus (Johnsen)

Johnsen [1991] esitti satunnaisia binääripuita generoivan algoritmin, jossa käytetty koodausmenetelmä voidaan kuvata seuraavasti. Oletetaan lehtisolmut sisältävä binääripuu, jossa on yksi sisäsolmu. Lisätään sisäsolmuja (lehtisolmuineen) esijärjestyksessä. Jotta lopputuloksena olisi haluttu puu, lisäykset voidaan tehdä vain yhdellä tavalla. Ennen kutakin lisäystä numeroidaan kaikki ne lehtisolmut, joiden paikalle esijärjestystä noudattaen olisi mahdollista lisätä sisäsolmu, kun ei välitetä siitä, mikä on lopputuloksena saatava puu. Numerointi tehdään nollasta alkaen niin ikään esijärjestyksessä (kuva 10). Koodisanan seuraavan alkion arvoksi tulee sen lehtisolmun

numero, jonka paikalle uusi sisäsolmu sitten lisätään. Johnsen nimitti tällaista lisäysoperaatiota ‘oksastukseksi’ (grafting).



Kuva 10. Mahdolliset uuden solmun lisäyskohdat (0, 1 ja 2), kun lisäykset tehdään esijärjestyksessä.

Kun n on satunnaisen binääripuun solmujen lukumäärä, Johnsenin algoritmi arpoo järjestyksessä n-1 koodisanan numeroa siten, että kaikki mahdolliset lopputuloksena saatavat koodisanat ovat yhtä todennäköisiä.

Yhden lisäysoperaation yhteydessä numeroitavia lehtisolmuja Johnsen nimitti ‘loppulehtisolmuiksi’ (trailing leaf), koska niitä ovat aina esijärjestyksessä viimeisen sisäsolmun jälkeen tulevat lehtisolmut; niiden lukumäärää binääripuussa t_i merkitsemme $tl(t_i)$:llä, joten niiden numerot ovat välillä $0...tl(t_i)-1$. Lisätystä sisäsolmusta tulee aina esijärjestyksessä viimeinen sisäsolmu. Kun teemme lisäysoperaation puuhun t_i, merkitsemme tuloksena saatua puuta t_{i+1}:llä ja lisäysoperaation määräämää viimeisintä koodisanan alkiota x_i:llä. Nyt esijärjestyksen ja sovellettavan koodausmenetelmän ominaisuuksien perusteella pätee

(5.20) $$tl(t_{i+1}) = tl(t_i)+1-x_i.$$

Merkitään L(m,j):llä niiden binääripuiden lukumäärää, jotka ovat rakennettavissa j lisäysoperaatiolla binääripuusta, jossa on m loppulehtisolmua. Johnsenin perustavaa laatua oleva tulos on, että tämä lukumäärä riippuu ainoastaan m:stä ja j:stä ja että L(m,j), $m\geq 2$, $j\geq 0$, voidaan laskea yhtälöistä

(5.21)
$$\begin{aligned} L(m,0) &= 1 \\ L(m,1) &= m \\ L(2,j) &= C_{j+1} \\ L(3,j) &= L(2, j+1) - L(2, j) \\ L(m,j) &= L(m-1, j+1) - L(m-2, j+1), \quad m>3. \end{aligned}$$

Kaavojen 5.20 ja 5.21 lukumäärien perusteella laskettavien todennäköisyyksien avulla voidaan arpoa generoitavan koodisanan numerot. Yhdellä lisäysoperaatiolla

puuhun t_i, missä i on puun solmujen lukumäärä, voidaan tuottaa $tl(t_i)=m$ erilaista puuta t_{i+1}. Lähtien puusta t_i erilaisten mahdollisten n-solmuisten puiden lukumäärä on L(m,n-i). Tämä sama lukumäärä on välttämättä myös tuloksena, kun lasketaan yhteen erilaisten mahdollisten n-solmuisten puiden lukumäärät lähtien jokaisesta m erilaisesta puusta t_{i+1}; toisin sanoen summataan yhteen $L(tl(t_{i+1}),n-i-1)$ laskettuna kullekin puulle t_{i+1}. Näiden lukumäärien avulla voidaan tarvittavat todennäköisyydet laskea satunnaisen koodisanan generoimiseksi algoritmin 5.22 mukaisesti.

```
(1)   procedure RandomBinaryTree
(2)   comment  Johnsen [1991]: satunnainen koodisana  n-alkioiselle binääripuulle;
(3)   integer array  x[0...n-1]        // koodisana rakentuu positioihin 1...n-1
(4)   x[0] := 0;  tl := 1              // alkuarvot
(5)   for  i := 1, n-1                 // i on nykyisen puun solmulukumäärä
(6)         tl := tl + 1- x[i-1]       // loppulehtisolmujen lkm;  kaava 5.20
(7)         valitaan satunnainen koodi k∈ [0,tl-1], missä kunkin k:n
(8)         todennäköisyys on L(tl+1-k, n-i-1) / L(tl, n-i)
(9)         x[i] := k                  // seuraava koodisanan alkio
(10)  end for
(11)  end procedure
```

Algoritmi 5.22. Satunnaisen binääripuun generointi Johnsenin algoritmilla.

Algoritmin 5.22 riveillä 7 - 8 esitetty puun t_{i+1} valinta puun t_i perusteella voidaan tehdä esim. seuraavasti: tuotetaan satunnaisluku väliltä [0...1), jonka jälkeen kumulatiivisesti summataan rivin 8 todennäköisyyksiä kunnes summa ylittää satunnaisluvun. Johnsen laskee keskimääräiseksi summausten määräksi vähemmän kuin kaksi yhtä koodisanan alkiota kohti.

Algoritmin ajoaikaisen tehokkuuden kannalta pitäisi kaikki mahdolliset n(n+1)/2 funktion L arvoa laskea ja taulukoida etukäteen. Taulukointivaiheen aika- ja tilavaatimukset ovat $O(n^2)$. Taulukoiden alkioiden arvot voidaan laskea soveltaen dynaamista ohjelmointia. Valmista taulukkoa hyödyntäen algoritmi toimii lineaarisessa ajassa, koska for-silmukan suorituskertojen lukumäärä on O(n) ja kullakin kierroksella suoritettava kumulatiivinen todennäköisyyksien summaus keskimäärin O(1).

Algoritmin ensimmäisen kerran suorittama funktion kutsu L(tl,n-i)=L(2,n-1) laskee erilaisten mahdollisten binääripuiden lukumäärän, kun tehdään n-1 lisäysoperaatiota lähtien puun juurisolmusta. Tämä luku on sama kuin kaikkien n-solmuisten binääripuiden lukumäärä C_n, joten Johnsenin algoritmi käyttää eksponentiaalisia lukuja.

Mäkinen [1999] osoitti, että on olemassa algoritmi, joka generoi Johnsenin koodausmenetelmän mukaisia sanoja lineaarisessa ajassa käyttäen lukuja, jotka ovat enintään $O(n^2)$. Tarkastelemme tätä seuraavassa, ja jaamme päättelyn neljään osaan.

1) Lucas *et al.* [1993] osoittivat, että kun binääripuiden solmujen lukumäärä on n, seuraavien koodausmenetelmien sanojen välillä voidaan tehdä ajassa O(n) yksikäsitteinen muunnos, joka perustuu pelkästään koodausten eikä niitä vastaavien puiden ominaisuuksiin: Zaks [1980], Zerling [1985], Pallo [1986], Mäkinen [1987]. Muunnoksessa tarvittavien lukujen suuruus on $O(n^2)$.

2) Martinin ja Orrin [1990] (käännöstaulu- eli oikean etäisyyden) menetelmässä käytetty solmujen numerointi on peilikuva Mäkisen [1987] (vasemman etäisyyden) menetelmän numeroinnista; muuten koodaukset ovat identtiset. Täten Martinin ja Orrin algoritmin aikavaatimuksen ja sen käyttämien lukujen suuruuden perusteella on olemassa algoritmi, joka generoi satunnaisia vasemman etäisyyden koodisanoja ajassa O(n) käyttäen lukuja, jotka ovat enintään $O(n^2)$. Lucasin *et al.* muunnoksen perusteella sama voidaan laajentaa koskemaan myös Zaksin, Zerlingin ja Pallon koodausmenetelmiä.

3) Seuraavaksi on tarkasteltava Zerlingin ja Johnsenin todistamia välttämättömiä ja riittäviä ehtoja, jotka pätevät koodisanojen alkioille. Zerlingin koodisana on muotoa $(x_{n-1},x_{n-2},...,x_1)$, missä ensimmäinen alkio x_n jätetään merkitsemättä, koska se on aina 0. Johnsenin koodisana on muotoa $(x_1,x_2,...,x_{n-1})$. Ehdot ovat, Zerling ensiksimainittuna,

$$0 \le x_i \le n-i-(x_{n-1}+x_{n-2}+...+x_{i+1}) \quad \text{ja}$$
$$0 \le x_i \le i-(x_1+x_2+...+x_{i-1}).$$

Kummassakin tapauksessa koodisanojen joukko on sama, vaikka samaa puuta vastaavat koodisanat eivät yleensä ole samat [Mäkinen, 1999]. Satunnaisten binääripuiden generoinnin kannalta tämä riittää, koska puiden järjestyksellä ei ole merkitystä; ratkaisevaa on se, että kutakin puuta vastaa yksikäsitteinen koodisana ja päinvastoin.

4) Koska kohdan 2) nojalla on olemassa algoritmi, joka generoi satunnaisia Zerlingin koodisanoja lineaarisessa ajassa käyttäen lukuja, jotka ovat enintään $O(n^2)$, kohdan 3) nojalla tämä sama algoritmi generoi satunnaisia Johnsenin koodisanoja lineaarisessa ajassa käyttäen lukuja, jotka ovat enintään $O(n^2)$.

5.8. Esijärjestys-numeroparit (Korsh)

Korshin [1993] koodausmenetelmässä jokainen binääripuun solmu esitetään numeroparina, jossa kukin numero voi olla joko nolla tai yksi. Parin ensimmäinen numero on ykkönen, jos solmulla on vasen lapsi; muutoin se on nolla. Parin jälkimmäinen numero on ykkönen, jos solmulla on oikea lapsi; muutoin se on nolla. Numeroparit eli solmut listataan esijärjestyksessä. Esimerkiksi koodaus (1,1),(0,1),(0,0),(0,0) esittää binääripuuta, jonka juurisolmulla on molemmat lapset ja juurisolmun vasemmalla lapsella on oikea lapsi. Koska n-solmuisessa binääripuussa on aina n-1 särmää ja koodauksen pituus on aina 2n numeroa, koodaus sisältää aina n-1 ykköstä ja n+1 nollaa.

Olkoon c mielivaltaista n-solmuista binääripuuta edustava koodaus. Jonon c *k-rotaatio* saadaan siirtämällä jonon k-1 ensimmäistä numeroparia jonon loppuun. Jonon c 1-rotaatio on siis jono itse. Jos puun solmut numeroidaan esijärjestyksessä 1...n, niin k-rotaatio on jono, joka saadaan aloittamalla koodisanan muodostaminen solmusta k, etenemällä solmuun n, palaamalla solmuun 1 ja etenemällä solmuun k-1.

Korsh osoitti seuraavat asiat: 1) Mielivaltaisen binääripuun T koodaus c ja sen kaikki k-rotaatiot voidaan tuottaa ainoastaan T:n perusteella eikä minkään muun binääripuun perusteella. 2) Kaikki jonon c k-rotaatiot, missä k=1...n, ovat erilaisia. 3) Yksikään jonon c k-rotaatio, missä k=2...n, ei edusta kokonaista n-solmuista binääripuuta. 4) Numerojono, joka koostuu mielivaltaisessa järjestyksessä n-1 ykkösestä ja n+1 nollasta, on aina joko n-solmuisen binääripuun koodaus tai sen k-rotaatio. Tällaisten jonojen lukumäärä on $\binom{2n}{n-1}$. Huomaa, että laskutoimitus n*C_n antaa tulokseksi saman luvun.

Korshin satunnaisia binääripuita generoiva algoritmi toimii seuraavasti. Generoidaan kohdan 4) mukainen satunnainen numerojono (esim. Atkinsonin ja Sackin [1992] algoritmin yhteydessä kuvatulla tavalla). Etsitään n erilaisesta mahdollisuudesta se k-rotaatio, joka edustaa binääripuuta. Tämän suorittamiseksi Korsh antaa algoritmin, joka kuvataan seuraavassa.

Olkoon d kohdan 4) mukainen satunnainen numerojono. Etsitään d:n lyhin alkuosa, jossa nollien lukumäärä ylittää ykkösten lukumäärän kahdella. Jos tämä alkuosa on erisuuri kuin d, siirretään alkuosa d:n loppuun ja aloitetaan alusta. Huomaa, että tämän algoritmin toteutuksessa d:n osamerkkijonojen siirtely voidaan välttää käyttämällä rengasmaista tietorakennetta.

Koska sekä kohdan 4) mukaisen numerojonon d generointi että d:n muuntaminen kelvolliseksi binääripuun koodaukseksi voidaan kumpikin toteuttaa lineaarisessa ajassa, menetelmällä voidaan generoida satunnaisia binääripuita lineaarisessa ajassa käyttäen lukuja, jotka ovat enintään 2n [Korsh, 1993].

5.9. Kielioppeihin perustuvat menetelmät

Tasapainotetut sulkujonot voidaan tuottaa kontekstittomalla kieliopilla, jossa on säännöt

(5.23) $$S \rightarrow (S)S \mid \lambda,$$

missä λ on tyhjä merkkijono ja kielioppeihin liittyvät merkinnät ja käsitteet samat kuin esim. Ahon ja Ullmanin [1972; 1995] teoksissa. Zaksin [1980] jonot voidaan tuottaa säännöillä

(5.24) $$S \rightarrow 1SS \mid 0.$$

Kun Zaksin jonosta poistetaan viimeinen nolla ja jonon jokainen ykkönen korvataan alkusululla ja nolla loppusululla, saadaan tasapainotettu sulkujono. Säännöt

$$S \rightarrow (A$$
$$A \rightarrow SA \mid)$$

tuottavat tasapainotettuja sulkujonoja (tyhjä jono mukaanlukien), joiden ympärillä on yksi ylimääräinen sulkupari [Hickey and Cohen, 1983].

Kielioppien 5.23 ja 5.24 sääntöjen valinnassa voidaan soveltaa todennäköisyys-kaavaa 5.13, kun halutaan generoida annetun pituisia satunnaisia kielen sanoja. Annettua binääripuuta vastaavat derivaatiopuut ovat samat generoitaessa Zaksin jono sääntöjen 5.24 mukaan ja generoitaessa tasapainotettu sulkujono sääntöjen 5.23 mukaan [Mäkinen, 1999]. Tästä seuraa, että tasapainotetuille sulkujonoille ilmoitetut resurssivaatimukset pätevät myös Zaksin jonoille.

Xiangin *et al.* [1997] koodausmenetelmässä kuhunkin binääripuun solmuun liitetään kirjain seuraavien sääntöjen mukaisesti: a, jos solmulla ei ole lapsia; b, jos solmulla on vain oikea lapsi; c, jos solmulla on molemmat lapset; d, jos solmulla on vain vasen lapsi. Kirjaimet listataan esijärjestyksessä. Binääripuita edustavat koodisanat voidaan tuottaa säännöillä

$$S \rightarrow a \mid bS \mid cSS \mid dS.$$

Samaa koodausmenetelmää noudattaen lehtisolmut sisältävät binääripuut voidaan tuottaa säännöillä

$$S \rightarrow a \mid cSS,$$

koska puussa on vain kahdenlaisia solmuja.

Kaikissa edelläesitetyissä koodausmenetelmissä koodisanoissa käytettävien merkkien joukko on riippumaton puun solmujen lukumäärästä. Monissa menetelmissä koodisanan alkiot ovat kokonaislukuja väliltä 1...n, missä n on puun solmujen lukumäärä. Vasemman etäisyyden menetelmälle kieliopin esitti Mäkinen [1987; 1998]. Jokaiselle koodisanan $(x_0,x_1,...,x_{n-1})$ alkiolle x_i pätee $x_0=0$ ja $0 \leq x_i \leq x_{i-1}+1$, missä i=1,...,n-1. Kun puun solmujen lukumäärä on enintään n, koodisanojen muodostama kieli on äärellinen ja täten säännöllinen kielioppi on riittävä. Säännöt ovat muotoa

$$X_{i,j} \rightarrow a_0X_{i+1,0} \mid a_1X_{i+1,1} \mid ... \mid a_{j+1}X_{i+1,j+1},$$

missä alaindeksipari i,j sääntöjen vasemmissa puolissa osoittaa koodisanan position (i) ja koodisanan alkion arvon i:nnessä positiossa (j); sääntöjen oikeissa puolissa positiota kasvatetaan yhdellä (i+1) ja koodisanan seuraavan alkion arvo voi olla välillä [0...j+1].

Hickey ja Cohen [1983] esittivät algoritmin, joka generoi satunnaisesti n:n pituisia mielivaltaisen yksiselitteisen (unambiguous) kontekstittoman kielen sanoja. Algoritmi on Arnoldin ja Sleepin [1980] menetelmän laajennus, joka laskee sovellettavien sääntöjen todennäköisyydet kussakin tilanteessa. Algoritmin aikavaatimus on $O((n \log n)^2)$ ja tilavaatimus O(n). Algoritmin aikavaatimusta optimoiva versio toimii ajassa O(n), mutta sitä ennen on konstruoitava taulukko, johon on laskettu todennäköisyydet kaikkia mahdollisesti eteentulevia tilanteita varten; tällöin algoritmin tilavaatimus on $O(n^{q+1})$, missä q on kieliopin apusymbolien (nonterminal) lukumäärä.

Mairson [1994] paransi Hickeyn ja Cohenin saavuttamia aika- ja tilavaatimuksia. Mairsonin algoritmissa ne ovat $O(n^2)$ ja O(n), ja aikavaatimusta optimoivassa versiossa O(n) ja $O(n^2)$. Algoritmi käyttää eksponentiaalisia lukuja [Mäkinen, 1999].

Hickey ja Cohen [1983] esittivät myös avoimen ongelman, joka voidaan muotoilla seuraavasti: mikä on se Chomskyn kielihierarkian kielten osaperhe, jossa on mahdollista tuottaa annetun pituisia satunnaisia kielen sanoja lineaarisessa ajassa ja tilassa? Hickey ja Cohen osoittivat, että kaikki säännölliset kielet kuuluvat tähän joukkoon. Edellä esitettyjen algoritmien resurssivaatimusten perusteella voidaan arvata,

etta kaikki kontekstittomat kielet eivät kuulu mainittuun joukkoon. Tasapainotetut sulkujonot ovat esimerkki ei-säännöllisestä kielestä, joka kuuluu mainittuun joukkoon.

5.10. Puut, joiden solmujen asteet määrätty (Atkinson)

Atkinsonin [1993] menetelmällä voidaan generoida n-solmuisia satunnaisia puita siten, että ennalta määritellään sellaisten solmujen lukumäärät d_i, joiden aste eli lasten lukumäärä on i, kun i=0...n-1. Olkoon $(d_0,d_1,...d_{n-1})$ tällaisen puun *tyyppivektori*. Lehtisolmut sisältävän binääripuun tyyppivektori on (t+1,0,t), kun loppunollat jätetään merkitsemättä ja n=2t+1. Tyyppivektorille pätee ehdon $\sum_{0\le i\le n-1} d_i = n$ lisäksi myös

(5.25) $$\sum_{0\le i\le n-1}(i-1)d_i = -1,$$

joka ilmaisee sen, että puun solmujen lukumäärä ylittää särmien lukumäärän yhdellä (vrt. Lause 2.5).

Puu voidaan koodata listaamalla puun solmujen asteet esijärjestyksessä. Kutsutaan tällaista listaa puun *astejonoksi* $g=g_1,g_2,...,g_n$: se alkaa luvulla g_1, jota seuraa g_1 astejonoa. Astejonolle välttämättömät ja riittävät ehdot ovat

(5.26) $$\sum_{1\le i\le n}(g_i-1) = -1 \quad \text{ja}$$

(5.27) $$\sum_{1\le i\le m}(g_i-1) \ge 0, \quad 0\le m<n.$$

Ehto 5.26 on ehto 5.25 uudelleenmuotoiltuna ja ehto 5.27 on sukua tasapainotettujen sulkujonojen ominaisuudelle, jossa loppusulkujen lukumäärä ei koskaan ylitä alkusulkujen lukumäärää.

Atkinsonin generointialgoritmi perustuu Raneyn v. 1960 todistamaan teoreemaan, joka voidaan muotoilla seuraavasti [Atkinson, 1993]. Oletetaan mielivaltainen kokonaislukujono $f=f_1,f_2,...,f_n$, jolle pätevät seuraavat ehdot:

i) $f_i \in [0,n-1]$, i=1...n;

ii) jonossa $d=(d_0,d_1,...d_{n-1})$ kukin d_i ilmoittaa numeroiden i lukumäärän jonossa f;

iii) jonolle d pätee ehto 5.25.

Jos f täyttää ehdot i), ii) ja iii), niin on olemassa yksi ja vain yksi jonon f rotaatio g (vrt. Korshin [1993] menetelmä), joka on astejono eli täyttää ehdot 5.26 ja 5.27 [Atkinson, 1993].

Kun ehdon 5.25 täyttävä tyyppivektori $d=(d_0,d_1,...d_{n-1})$ on annettu, satunnainen puu generoidaan seuraavasti: 1) Muodostetaan jono, jonka d_0 ensimmäistä alkiota ovat nollia, d_1 seuraavaa ykkösiä jne., ja luodaan tämän jonon satunnainen permutaatio f [esim. Knuth, 1969; Reingold *et al.*, 1977]; f täyttää nyt ehdot i), ii) ja iii). 2) Jonosta f etsitään astejonon alkupositio algoritmilla 5.28, joka etsii pienimmän osasumman $\sum_{1\le i\le m} f_i'$, missä $f_i'=f_i-1$. 3) Astejono g saadaan tekemällä jonoon f rotaatio, jonka seurauksena mainittu alkupositio tulee ensimmäiseksi.

```
procedure FindPosition(f)
comment Atkinson [1993]: astejonon alkuposition haku;
          input-jono f täyttää ehdot i)-iii);
min := 0;  sum := 0
for i:=1 to n do
        sum := sum + f_i - 1
        if (sum<min) then min := sum; k := i end if
end for
return  k mod (n+1)
end procedure
```

Algoritmi 5.28. Astejonon alkuposition haku.

5.11. Puut abstraktin verkon erikoistapauksena; Vapaa puu

Verkkoteoriassa puu tavallisesti määritellään abstraktin verkon erikoistapauksena. Tämän havainnollistamiseksi esitellään seuraavassa verkon määritelmä ja termit soveltuvassa muodossa. Täydellisemmät määritelmät ovat löydettävissä verkkoteorian oppikirjoista.

Abstrakti verkko eli *graafi* (graph) muodostuu joukosta *solmuja* ja joukosta solmuja yhdistäviä *särmiä*. Särmät tulkitsemme tässä kaksisuuntaisiksi. Solmun v *kokonaisaste* on niiden särmien lukumäärä, jotka yhdistävät solmun v johonkin verkon solmuun tai solmuihin. Solmujen v_1 ja v_k välinen *polku* (path) on solmujono $v_1,v_2,...,v_k$, missä jokaista kahta solmua v_i ja v_{i+1} yhdistää särmä. Verkko on *yhtenäinen* (connected), jos verkon minkä tahansa kahden solmun välillä on polku. Nyt n-solmuinen verkko on *puu*, jos se on yhtenäinen ja särmien lukumäärä on n-1 [esim. Knuth, 1973a, s. 362]. Huomaa, että puun juurta ei ole määritelty. Käytämme tällaisesta puusta nimitystä *vapaa puu* (free tree). Alonso ja Schott [1995, s. 25] kutsuvat sitä nimellä Cayleyn puu (Cayley arborescence). Kaikki tässä kohdassa tarkasteltavat puut ovat yllämainitun määritelmän mukaisia.

Esitetään seuraavassa klassinen algoritmi (Prüfer, v. 1918), jonka avulla voidaan konstruoida n-solmuinen numeroidut solmut sisältävä vapaa puu n-2 -alkioisen listan $a=(a_1,a_2,...,a_{n-2})$, $1 \le a_i \le n$, perusteella [Nijenhuis and Wilf, 1975, p. 203; Quiroz, 1989]. Kutsumme tällaista listaa Prüferin listaksi. Mainittujen puiden lukumäärä on n^{n-2} [esim. Knuth, 1973a, Ex. 2.3.4.4-22]. Erilaisten Prüferin listojen lukumäärä on selvästikin n^{n-2}. Muunnos on yksikäsitteinen myös toiseen suuntaan. Lista a annetaan algoritmille syöteparametrina. Alkutilanteessa 'puu' sisältää solmut 1...n, mutta niiden välillä ei ole yhtään särmää.

```
procedure FreeTree(a)
comment  Nijenhuis ja Wilf [1975]: vapaa puu (free tree) numeroiduilla solmuilla;
                lista a=(a1,a2,...an-2), 1≤ai≤n, on syöteparametri;
b := (1,2,...,n)
for i := 1 to n-2 do
        x := pienin numero, joka on jonossa a mutta ei ole jonossa b
        Yhdistä särmällä puun solmut ai ja x
        Poista ai jonosta a ja x jonosta b
end for
Yhdistä särmällä kaksi solmua, jotka jäljelläolevat kaksi jonon b numeroa ilmaisevat
end procedure
```

Algoritmi 5.29. Vapaan puun (free tree) konstruointi.

Nijenhuisin ja Wilfin [1975] algoritmin 5.29 esimerkkitoteutuksen aikavaatimus on $O(n^2)$ [Quiroz, 1989], mutta Quiroz esitti samalle algoritmille toteutuksen, jonka aikavaatimus on O(n). Koska satunnainen lista a voidaan generoida ajassa O(n), satunnaisia puita voidaan generoida algoritmilla 5.29 ajassa O(n).

5.12. Vapaa puu, solmujen kokonaisasteet määrätty

Algoritmin 5.29 avulla voidaan generoida satunnaisia numeroituja vapaita puita siten, että ennalta määritellään niiden solmujen lukumäärät d_i, joiden kokonaisaste on i, kun i=1...n-1 (vrt. Atkinsonin [1993] menetelmä). Sovellamme algoritmia *t-puiden* (t-ary tree) generointiin. Kutsumme (määritelmän 2.1 mukaiseksi) t-puuksi sellaista lehti-solmut sisältävää puuta, jonka jokaisen sisäsolmun aste on t. Vastaavien vapaiden puiden solmujen kokonaisasteita koskevien lainalaisuuksien selvittämiseksi tarkastelemme ensin tavallisia puita.

Olkoon t-puun kaikkien solmujen lukumäärä n ja sisäsolmujen lukumäärä k. Yleistämällä binääripuita koskeva Lauseen 2.5 päättely t-puihin, lehtisolmujen luku-

määräksi saadaan tk-(k-1). Täten t-puun solmujen lukumäärille pätee n=tk+1.

Sovellamme näitä tuloksia vapaisiin puihin: n-solmuisessa t-puuta vastaavassa vapaassa puussa on oltava yksi solmu (juuri), jonka kokonaisaste on t, k-1 solmua (sisäsolmut), joiden kokonaisaste on t+1 ja tk-(k-1) solmua (lehdet), joiden kokonaisaste on 1. Käytämme tällaisista puista nimitystä *vapaa t-puu*.

Seuraava lause ilmaisee vapaan puun solmujen kokonaisasteiden ja vastaavan Prüferin listan välisen yhteyden:

Lause 5.30. [Quiroz, 1989] Vapaan puun solmun, jonka numero on i, kokonaisaste on x+1, missä x on numeron i esiintymien lukumäärä vastaavassa Prüferin listassa.

Jos lehtisolmut sisältävässä vapaassa t-puussa on n=tk+1 numeroitua solmua, sitä vastaavassa Prüferin listassa on yksi numero, joka esiintyy listassa t-1 kertaa ja k-1 numeroa, jotka esiintyvät listassa t kertaa. Listan pituudeksi tulee tk-1. Satunnaisia puita voidaan nyt generoida algoritmin 5.31 avulla.

procedure T-aryFreeTree
comment Quiroz [1989]: vapaa t-puu numeroiduilla solmuilla;
N := {1,2,...,tk+1}
M := {m_1, m_2, ..., m_k}, missä M on joukon N satunnainen osajoukko (ei toistoja)
a := (m_1,...,m_1,m_2,...,m_2,...,m_k,...,m_k), missä kukin m_1...m_{k-1} esiintyy
t kertaa ja m_k t-1 kertaa
a := a:n satunnainen permutaatio
FreeTree(a)
end procedure

Algoritmi 5.31. Vapaan t-puun konstruointi.

Satunnaisen osajoukon ja satunnaisen permutaation generoinnin aikavaatimukset ovat O(n) [esim. Knuth, 1969; Reingold *et al.*, 1977]. Koska kutsun FreeTree(a) aikavaatimus on O(n) (ks. kohta 5.11), algoritmin 5.31 aikavaatimus on O(n).

Quiroz [1989] esitti myös algoritmia 5.29 hyödyntävät algoritmit, jotka generoivat vapaita t-puita, joissa vain lehdet on numeroitu (terminally labeled t-ary tree), sekä puita, joissa kutakin kokonaisastetta olevien solmujen lukumäärät on annettu.

5.13. Satunnainen etsintäpuu

Tässä tutkielmassa binääripuita tarkastellaan kombinatorisesta näkökulmasta: jokaisen erilaisen n-solmuisen puun *a priori* -todennäköisyys on $1/C_n$. Puun todennäköisyys puun muodon funktiona voidaan määritellä myös toisin. Tässä kohdassa määritellään *satunnaisesti muodostettu binäärinen etsintäpuu* (random binary search tree) eli *satunnainen etsintäpuu*, joka yhdessä tasajakaumaa noudattavan binääripuun eli tasajakaumapuun kanssa on tärkein binääripuiden todennäköisyysmalli. Lisäksi esitellään annetun etsintäpuun todennäköisyyden laskentakaavat, ja lopussa on luetteloitu kirjallisuudessa esitettyjä satunnaisten etsintäpuiden generointialgoritmeja. Satunnaisten tasajakaumapuiden ja etsintäpuiden asymptoottisia ominaisuuksia vertaillaan luvussa 6.

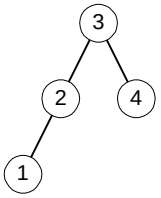
Määritelmä 5.32. Oletetaan binääripuu (ilman lehtisolmuja), jonka solmut on numeroitu 1:stä n:ään. Puu on *etsintäpuu* (search tree), jos sen jokaiselle solmulle s pätee 1) s:n vasemman alipuun jokaisen solmun numero on pienempi kuin s:n numero, ja 2) s:n oikean alipuun jokaisen solmun numero on suurempi kuin s:n numero.

Etsintäpuuhun kohdistuvat haku- ja lisäysoperaatiot voidaan tehdä algoritmin 5.33 avulla, missä ‘solmu’ on lisättävä/haettava solmu ja ‘juuri’ on (ali)puun juurisolmu tai tyhjä. Lisäys- ja hakuoperaatiot ovat analogisia ja ne on yhdistetty yhdeksi algoritmiksi. Kahden solmun vertailuoperaatiot vertaavat solmujen numeroita keskenään.

```
procedure SearchInsert(solmu,juuri)
if  (juuri  on tyhjä) then solmu ei ole puussa / lisätään solmu tähän kohtaan puussa
else if  (solmu<juuri) then SearchInsert(solmu, juuren vasen lapsi)
else if  (solmu>juuri) then SearchInsert(solmu, juuren oikea lapsi)
else solmu on jo puussa
end if
end procedure
```

Algoritmi 5.33. Etsintäpuun yhdistetty haku- ja lisäysalgoritmi.

Satunnainen etsintäpuu muodostetaan lisäämällä tyhjään etsintäpuuhun satunnaisessa järjestyksessä solmut 1...n algoritmin 5.33 avulla. Solmujen numerot lisäysjärjestyksessä lueteltuna muodostavat siis satunnaisen numeroista 1...n koostuvan permutaation, missä jokaisen permutaation todennäköisyys on 1/n!.



Kuva 11. Permutaatioita 3214, 3241 ja 3421 vastaava etsintäpuu.

Kun tarkastelemme puun muotoa, yksi etsintäpuu listamaisia puita lukuunottamatta voi olla tuloksena useammasta kuin yhdestä eri permutaatiosta (kuva 11). Knuth [1973b, Ex. 6.2.2-5] huomautti, että annetun puun generoivien permutaatioiden lukumäärä on a*b*c, missä a (vast. b) on niiden erilaisten permutaatioiden lukumäärä, jotka generoivat puun vasemman (vast. oikean) alipuun ja c ilmaisee, monellako eri tavalla lisäykset voidaan ‘lomittaa’ keskenään.

Seuraavassa esitettävien rekursiivisten laskentakaavojen [esim. Casas *et al.*, 1991; Baeza-Yates *et al.*, 1992] avulla voidaan laskea periaatteessa mielivaltaisen etsintäpuun todennäköisyys. Olkoot binääripuun T vasen ja oikea alipuu Tl ja Tr, T:n solmujen lukumäärä $|T|$, puun T generoivien permutaatioiden lukumäärä N(T) ja puun T todennäköisyys p(T). Kun $|T|=0$, on N(T)=1; muulloin

$$N(T) = N(Tl)\, N(Tr) \frac{(|T|-1)!}{|Tl|!\ |Tr|!}.$$

Kun $|T|=0$, on p(T)=1; muulloin

$$p(T) = N(T) / |T|! = p(Tl)\, p(Tr) / |T|.$$

Esimerkiksi 7-solmuinen täysin tasapainoinen etsintäpuu voidaan tuottaa 80 eri permutaatiolla (esim. 4,2,1,3,6,5,7) 7!=5040 erilaisesta mahdollisuudesta, kun taas erilaiset listamaiset puut ovat kukin tuotettavissa vain yhdellä permutaatiolla.

Kohdassa 3.3 tarkasteltiin *puupermutaatioita*, jotka ovat permutaatioiden osajoukko. Samassa kohdassa annettu binääripuun konstruointimenetelmä vaatii syötteekseen puupermutaation, toisin kuin algoritmi 5.33. Kaikista saman permutaatiopuun generoivista permutaatioista puupermutaatio on se, joka on aakkosjärjestyksessä pienin, ja sen perusteella puu rakentuu esijärjestyksessä [Knott, 1977].

Satunnaisen etsintäpuun generoinnin keskimääräinen aikavaatimus algoritmin 5.33 avulla on O(n log n) [esim. Devroye and Robson, 1995; Sedgewick and Flajolet, 1996]. Etsintäpuu on kuitenkin mahdollista konstruoida lineaarisessa ajassa, jos syötteenä

annettu permutaatio on puupermutaatio: Er [1987] esitti tätä varten algoritmin. Devroyen and Robsonin [1995] lineaarinen binääripuun koodausta soveltava algoritmi ei käsittele puupermutaatioita, vaan lisää solmuja suoraan puun lehtien paikalle. Lineaarisessa ajassa toimiva etsintäpuita generoiva algoritmi saadaan myös muuttamalla algoritmissa 5.1 sovellettavia todennäköisyyksiä: jos kunkin k:n todennäköisyys on C_kC_{n-1-k}/C_n, missä $k \in [0,n-1]$, algoritmi generoi satunnaisia tasajakaumapuita; jos taas jokainen k on yhtä todennäköinen, algoritmi generoi satunnaisia etsintäpuita [Sprugnoli, 1992].

5.14. Kirjallisuuskatsaus ja sovelluksia

Algoritmeja, joita suorasti tai epäsuorasti voidaan soveltaa erilaisten satunnaisten puiden generointiin, on 1970-luvun jälkipuoliskolta lähtien julkaistu monia. Tässä kohdassa esitetään luettelonomainen katsaus satunnaisia puita käsittelevään kirjallisuuteen ja artikkeleihin siltä osin kuin niitä ei edellä ole käsitelty. Luettelossa esitettyjä käsitteitä ja termejä ei välttämättä määritellä.

Satunnaisten puiden generointi:

- käänteisiä numerointialgoritmeja sekä yleisiä puiden generointimenetelmiä käsiteltiin lyhyesti kohdassa 5.1
- Nijenhuis ja Wilf [1975] esittivät hakuteoksessaan mm. algoritmit, jotka generoivat satunnaisia vapaita puita (free tree): algoritmi RANTRE generoi puita, joiden solmut on numeroitu, ja algoritmi RANRUT juurellisia puita, joiden solmut ovat numeroimattomia
- generointialgoritmi satunnaisille vapaille puille, solmut numeroimattomia, juurta ei määritelty [Wilf, 1981]
- generointialgoritmi järjestämättömille (unordered) satunnaisille binääripuille [Furnas, 1984]
- Dershowitz ja Zaks [1989] kehittivät puihin liittyvän mallin (pattern) käsitteen, jossa puun eri osat kuvataan erilaisten symbolien avulla; tähän perustuen Alonso *et al.* [1997] (ks. myös Alonso ja Schott [1995]) esittivät menetelmän useiden erilaisten satunnaisten puiden generointiin

- generointialgoritmi satunnaisille t-puille (t-ary tree) käyttäen koodauksena käännöstauluja [Martin, 1991]
- generointialgoritmi satunnaisille t-puille, joiden korkeus on määrätty [Atkinson and Sack, 1994]
- generointialgoritmi satunnaisille Schröderin puille (Schröder tree) [Alonso *et al.*, 1997b]
- yleiskatsaus satunnaisten binääripuiden generointiin [Mäkinen, 1999]
- Alonso ja Schott [1995] lienee laajin julkaistu algoritmikokoelma, joka keskittyy yksinomaan erityyppisten puiden satunnaiseen generointiin; teoksen aihepiiriin kuuluvat muun muassa
  - -- kasvavat puut (increasing trees): solmut on numeroitu siten, että jokainen polku juuresta lehteen on solmunumeron suhteen kasvava
  - -- Rémyn algoritmi
  - -- vapaat puut (Cayley arborescences/free trees)
  - -- 1-1 -vastaavuudet puiden ja Dyckin sanojen (Dyck word) sekä Motzkinin sanojen (Motzkin word) välillä (ks. myös Alonso [1994])
  - -- mallit (patterns)
  - -- värilliset puut (colored trees) (ks. myös Alonso ja Schott [1995b])
  - -- unaari-binääripuut (unary-binary trees)
  - -- vinot puut (skew trees) (ks. myös Barcucci *et al.* [1994])
  - -- Nijenhuisin ja Wilfin [1975] RANRUT -algoritmin analyysi sekä yksikkö- että logaritmista kustannusperiaatetta soveltaen
  - -- binääripuut, joiden korkeus on määrätty
  - -- rinnakkaisalgoritmien soveltaminen Dyckin ja Motzkinin sanojen generointiin.

Muita puihin liittyviä artikkeleita:

- yleiskatsaus (binääri)puiden esiintymiseen ja sovelluksiin eri tieteissä; binääripuiden visualisointi luonnollisten puiden kaltaisina [Viennot, 1990]
- satunnaisten binääripuiden soveltaminen lohkorakenteisten ohjelmointikielten

kääntäjien testaukseen: generoidaan satunnaisia tasapainotettuja sulkujonoja, jotka muunnetaan kielen sisäkkäisiksi/peräkkäisiksi lohkorakenteiksi [Arnold ja Sleep, 1980]

- Monte Carlo -simulaatiot ja satunnaiset puut [Furnas, 1984]
- satunnaisten aritmeettisten lausekkeiden generointi [Bainbridge, 1992]
- binääripuun rakenteen analysointi puun Hortonin-Strahlerin luvun avulla [Viennot, 1990; Devroye and Kruszewski, 1994]
- satunnaisten binääripuiden visualisointi luonnollisten puiden kaltaisina Postscript-tulostusohjelmiston avulla; piirtämisprosessiin liittyvät parametrit, joilla tulostuvan puun ulkonäköä voidaan säädellä [Devroye and Kruszewski, 1995]
- geneettisessä ohjelmoinnissa sovellettavat satunnaiset puut [Iba, 1996]
- Java-kielinen lähdekoodi ohjelmalle, joka generoi ja piirtää näytölle satunnaisen binääripuun [Goodrich and Tamassia, 1998].

6. Binääripuiden ominaisuuksia

Tässä luvussa esitetään todistus binääripuiden lukumäärän kaavalle sekä vertaillaan satunnaisen tasajakaumapuun ja satunnaisen etsintäpuun asymptoottisia ominaisuuksia erilaisten tunnuslukujen avulla.

6.1. Binääripuiden lukumäärän ja Catalanin luvun välinen yhteys

Tässä kohdassa esitetään todistus sille, että erilaisten n-solmuisten binääripuiden lukumäärä b_n on sama kuin n:s Catalanin luku C_n eli

$$b_n = C_n = \frac{1}{n+1}\binom{2n}{n}.$$

Muita binääripuiden ominaisuuksiin ja Catalanin lukuun liittyviä asioita tarkasteltiin jo luvussa 2.2.

Identiteetti $b_n = C_n$ voidaan todistaa useilla eri tavoilla, joista tunnetuin lienee rekursioyhtälön 5.2 ratkaiseminen generoivia funktioita soveltamalla [ks. esim. Knuth, 1973a]. Ko. yhtäsuuruus voidaan kuitenkin johtaa myös suoraan sulkujonojen ominaisuuksien perusteella. Binääripuiden ja tasapainotettujen sulkujonojen välinen 1-1 -vastaavuus on jo osoitettu kohdassa 3.2, joten riittää, että selvitämme erilaisten 2n-pituisten tasapainotettujen sulkujonojen lukumäärän.

Määritelmä 6.1. Olkoon $BP_{n,n}$ joukko, joka sisältää kaikki sellaiset tasapainotetut sulkujonot (balanced parentheses), joissa on n alkusulkua ja n loppusulkua, ja $NBP_{n,n}$ joukko, joka sisältää kaikki sellaiset ei-tasapainotetut sulkujonot, joissa on n alkusulkua ja n loppusulkua. Olkoon $P_{n-1,n+1}$ joukko, joka sisältää kaikki sellaiset sulkujonot, joissa on n-1 alkusulkua ja n+1 loppusulkua. Funktio card(X) palauttaa joukon X alkioiden lukumäärän.

Lause 6.2. [Even, 1979] On olemassa 1-1 -vastaavuus joukkojen $NBP_{n,n}$ ja $P_{n-1,n+1}$ välillä.

Todistus. i) Oletetaan mielivaltainen sulkujono $p=p_1p_2...p_{2n} \in NBP_{n,n}$ (vrt. kuva 9, polku RTK silloin kun piste R sijaitsee suoralla y=0). Etsitään p:n pienin alkuosa $p_1...p_j$, jossa loppusulkuja on yksi enemmän kuin alkusulkuja. Nyt loppuosassa $p_{j+1}...p_{2n}$ on alkusulkuja yksi enemmän kuin loppusulkuja. Käännetään kaikki loppuosan $p_{j+1}...p_{2n}$

merkit toisiksi, ts. alkusulut loppusuluiksi ja loppusulut alkusuluiksi. Nyt pätee $p \in P_{n-1,n+1}$, ja muunnos on yksikäsitteinen.

ii) Oletetaan mielivaltainen sulkujono $p=p_1p_2...p_{2n} \in P_{n-1,n+1}$. Etsitään p:n pienin alkuosa $p_1...p_j$, jossa loppusulkuja on yksi enemmän kuin alkusulkuja. Nyt loppuosassa $p_{j+1}...p_{2n}$ on loppusulkuja yksi enemmän kuin alkusulkuja. Käännetään kaikki loppuosan $p_{j+1}...p_{2n}$ merkit toisiksi. Nyt pätee $p \in NBP_{n,n}$, ja muunnos on yksikäsitteinen.

iii) Huomataan, että muunnokset i) ja ii) ovat käänteisiä toisilleen. Toisin sanoen, soveltamalla muunnosta i) mielivaltaiseen sulkujonoon $p \in NBP_{n,n}$, saamme jonon $q \in P_{n-1,n+1}$. Nyt jos sovellamme muunnosta ii) jonoon q, lopputuloksena saamme alkuperäisen jonon p. Sama pätee kääntäen: lähtien mielivaltaisesta sulkujonosta $q \in P_{n-1,n+1}$ saamme muunnoksen ii) avulla jonon $p \in NBP_{n,n}$, ja soveltamalla muunnosta i) jonoon p, saamme alkuperäisen jonon q. Täten joukkojen $NBP_{n,n}$ ja $P_{n-1,n+1}$ välillä on 1-1 -vastaavuus. ?

Lause 6.3. [Even, 1979] On voimassa $\mathrm{card}(NBP_{n,n}) = \mathrm{card}(P_{n-1,n+1}) = \binom{2n}{n-1}$.

Todistus. Väitteen vasemmanpuoleinen yhtäsuuruus pätee Lauseen 6.2 nojalla. Oikeanpuoleisen yhtäsuuruuden kohdalla on kyseessä kombinaatioiden lukumäärä: monellako tavalla n-1 alkusulkua voidaan sijoitella 2n positioon. ?

Lause 6.4. [Even, 1979] On voimassa $\mathrm{card}(BP_{n,n}) = \frac{1}{n+1}\binom{2n}{n}$.

Todistus. Sellainen sulkujono, jossa on n alkusulkua ja n loppusulkua, voi olla joko tasapainotettu tai ei-tasapainotettu. Siis pätee

$$\binom{2n}{n} = \mathrm{card}(BP_{n,n}) + \mathrm{card}(NBP_{n,n})$$

eli

$$\mathrm{card}(BP_{n,n}) = \binom{2n}{n} - \binom{2n}{n-1} = \frac{1}{n+1}\binom{2n}{n}.$$?

6.2. Satunnaisten binääripuiden asymptoottisia ominaisuuksia

Tässä kohdassa tarkastelemme satunnaisen etsintäpuun ja satunnaisen tasajakaumapuun rakenteeseen liittyviä keskimääräisiä eroja erilaisten tunnuslukujen avulla.

Yleistäen voidaan sanoa, että mitä tasapainoisempi etsintäpuu on, sitä todennäköisempi se on. Tätä etsintäpuun taipumusta tasapainoisuuteen luonnehdimme seuraavassa. Puun korkeutta h voidaan pitää yhtenä tapana mitata puun tasapainoisuutta. Korkeus ei kuitenkaan ole paras mahdollinen mittari, koska se määräytyy puun yhden polun perusteella. Parempi tunnusluku on puun *sisäinen polunpituus* (internal path length), joka saadaan laskemalla yhteen kaikkien sisäsolmujen etäisyydet juurisolmusta. Puun sisäsolmujen keskietäisyys d juurisolmusta on tällöin sama kuin puun sisäinen polunpituus jaettuna sisäsolmujen lukumäärällä. Etsintäpuilla näiden ominaisuuksien odotusarvot h_n [Devroye and Reed, 1995] ja d_n ovat

$$h_n = c \ln n + O(\log \log n) \approx 4.31107 \ln n, \quad \text{ja}$$
$$d_n = 2(1+1/n)(H_{n+1}-1) - 2 \approx 1.3863 \log_2 n - 2.8456,$$

missä vakio c on yhtälön $c \ln(2e/c)=1$ ratkaisu, ln on luonnollinen logaritmi, e tämän kantaluku ja $H_n = \sum_{i=1}^{n} 1/i$. Tasajakaumapuiden tapauksessa h_n [Flajolet and Odlyzko, 1982] ja d_n ovat

$$h_n = 2\sqrt{pn} + O(n^{1/4+\varepsilon}) \approx 3.5449\sqrt{n} \quad (\text{kaikilla } \varepsilon>0), \text{ ja}$$
$$d_n = (1+1/n)\, 4^n \binom{2n}{n}^{-1} - 3 - 1/n = \sqrt{pn} - 3 + O(n^{-0.5}) \approx 1.77245\sqrt{n} - 3$$

(ks. myös Knuth [1973b]; Gonnet and Baeza-Yates [1991]; Sedgewick and Flajolet [1996]). Etsintäpuun tasajakaumapuuta suurempi taipumus tasapainoisuuteen näkyy selvästi puun korkeutta kuvaavissa keskimääräisissä suureissa: etsintäpuilla ne ovat logaritmisia, kun taas tasajakaumapuilla ne ovat likimäärin verrannollisia solmujen lukumäärän neliöjuureen.

Koska tasajakaumapuu on etsintäpuuta keskimäärin korkeampi, se sisältää etsintäpuuta enemmän listamaisia osia. Mitä listamaisempi puu on, sitä enemmän se sisältää solmuja, joiden aste on yksi. Seuraavassa johdamme asymptoottiset arvot keskimääräisen tasajakaumapuun eri asteisten solmujen lukumäärille.

Määritelmä 6.5. Oletetaan n-solmuinen binääripuu T_n ilman lehtisolmuja. Olkoon $D_i(T_n)$ puun sellaisten solmujen lukumäärä, joiden aste on i. Olkoon $E\, D_i(T_n)$ n-solmuisten tasajakauma- tai etsintäpuiden i-asteisten solmujen lukumäärän odotusarvo.

Lause 6.6. [Sedgewick and Flajolet, 1996, p. 241] Tasajakaumapuille pätee $\lim_{n\to\infty} E\, D_1(T_n) = n/2$.

Todistus. Kun n>1, C_n erilaisesta n-solmuisesta tasajakaumapuusta C_{n-1} on sellaisia, joiden juuren vasen (vast. oikea) alipuu on tyhjä mutta oikea (vast. vasen) alipuu ei (ks. myös kohta 5.2). Todennäköisyys sille, että täsmälleen yksi juuren alipuu on tyhjä, on $2C_{n-1}/C_n$. Koska puun mielivaltainen solmu on tasajakaumapuun juuri, tätä päättelyä voidaan soveltaa puun mielivaltaiseen alipuuhun. Koska $\lim_{n\to\infty} 2C_{n-1}/C_n = 1/2$, niin lause seuraa. ?

Lemma 6.7. Oletetaan Määritelmän 6.5 mukainen binääripuu T_n, jolle pätee $D_1(T_n) = k$. Tällöin on $D_0(T_n) = (n+1-k)/2$ ja $D_2(T_n) = (n-1-k)/2$.

Todistus. Lauseen 2.5 nojalla puun T_n tyhjien alipuiden lukumäärä on n+1. Näistä k on sellaisissa solmuissa, joiden aste on 1. Täten n+1-k tyhjää alipuuta on sellaisissa solmuissa, joiden aste ei ole 1. Koska solmuilla, joiden aste on 2, ei ole tyhjiä alipuita, loput tyhjät alipuut ovat sellaisten solmujen alipuita, joiden aste on 0. Täten $D_0(T_n) = (n+1-k)/2$. Koska $D_0(T_n)+D_1(T_n)+D_2(T_n) = n$, on $D_2(T_n) = (n-1-k)/2$. ?

Lause 6.8. Tasajakaumapuille pätee

i) $\lim_{n\to\infty} E\, D_0(T_n) = n/4$,

ii) $\lim_{n\to\infty} E\, D_1(T_n) = n/2$ ja

iii) $\lim_{n\to\infty} E\, D_2(T_n) = n/4$.

Todistus. i) Lemman 6.7 nojalla on $D_0(T_n) = (n+1-D_1(T_n))/2$. Lauseen 6.6 ja Lemman 6.7 nojalla on $\lim_{n\to\infty} E\, D_0(T_n) = \lim_{n\to\infty}(n+1-n/2)/2 = n/4$. ii) Lause 6.6. iii) Todistetaan kuten kohta i). ?

Sedgewick ja Flajolet [1996, s. 254] todistavat Lauseen 6.8 kohdan i) soveltamalla generoivia funktioita. Lauseen 6.8 todistuksia ovat esittäneet ainakin Brinck ja Foo [1981] sekä Mahmoud [1995], ja niissä on sovellettu generoivia funktioita ja rekursioyhtälöitä.

Satunnaisten etsintäpuiden eri asteen omaavien solmujen lukumäärien asymptoottiset arvot antaa seuraava lause:

Lause 6.9. [Mahmoud, 1986] Satunnaisille etsintäpuille pätee $\lim_{n\to\infty} E\, D_i(T_n) = n/3+O(1)$, $i=0,1,2$.

Yhteenvetona toteamme, että riittävän suuren satunnaisen tasajakaumapuun solmuista keskimäärin noin neljäsosa on sellaisia, joilla ei ole lapsia, noin puolet sellaisia, joilla on yksi lapsi ja noin neljäsosa sellaisia, joilla on kaksi lasta. Etsintäpuilla vastaavat osuudet ovat noin kolmasosa kussakin tapauksessa (tapaus i=0: ks. myös esim. Sedgewick ja Flajolet [1996, s. 254]). Odotusarvot eivät kuitenkaan ole yhtäsuuret [Mahmoud, 1986].

7. Kuuden algoritmin tehokkuustutkimus

Tässä luvussa tutkitaan viiden aikavaatimukseltaan lineaarisen satunnaisia binääripuita generoivan algoritmin keskinäistä tehokkuutta. Algoritmit toteutetaan, algoritmien vertailussa sovellettavat kriteerit määritellään ja valittujen vertailukriteerien asymptoottiset arvot puun koon funktiona selvitetään ohjelmallisesti. Algoritmit ovat Arnold ja Sleep [1980], Martin ja Orr [1990], Atkinson ja Sack [1992], Korsh [1993] sekä Rémyn algoritmi vuodelta 1985 [Alonso and Schott, 1995]. Kuudes algoritmi on Johnsen [1991], jonka esitaulukointivaiheen aikavaatimus on $O(n^2)$, mutta generointi-vaiheen lineaarinen.

Jokaiselle algoritmille tehdään seuraavat asiat: 1) algoritmin toteutuksen oikeellisuuden verifiointi: tilastollisten menetelmien avulla todetaan generoitujen binääripuiden satunnaisuus; 2) binääripuiden satunnainen generointi algoritmin soveltaman koodauksen muodossa ja vertailukriteereiksi valittujen operaatioiden lukumäärien laskenta; 3) lopullisten binääripuiden konstruointi koodausten perusteella ja vertailukriteereiksi valittujen operaatioiden lukumäärien laskenta sekä 4) tulosten analysointi.

Lisäksi Alonson and Schottin [1995, Appendix 2, Alg. 2.1] toteutus Rémyn algoritmille osoitetaan virheelliseksi: sen tuottamien satunnaisten binääripuiden jakauma ei ole tasainen. Tätä tutkimusta varten Rémyn algoritmi on ohjelmoitu uudelleen Alonson ja Schottin toteutuksen periaatteita mukaillen.

7.1. Algoritmien kustannusten laskenta ja vertailukriteerit

Tässä kohdassa esitellään tutkittavien algoritmien vertailukriteerien valinnan perusteet sekä algoritmien kustannusten laskennassa käytetty menetelmä.

7.1.1. Teoria

Algoritmien analysoinnissa käytettävä O-merkintätapa ilmoittaa resurssivaatimuksen *suuruusluokan* ongelman koon funktiona. Kun tarkastelu halutaan tehdä tätä tarkemmalla tasolla, resurssivaatimusten mittaamiselle ei ole olemassa yhtenäistä ja yleispätevää käytäntöä johtuen algoritmien ja niiden kuvaustapojen (kuten myös

toteutusympäristöjen) erilaisuudesta. Seuraavassa pohdimme algoritmien O-lukua tarkemmalla tasolla tapahtuvaan kustannusten laskentaan liittyviä kysymyksiä.

Teoriassa kaikkien algoritmien kustannusten laskenta on mahdollista yhteismitallistaa käyttämällä perustana jotakin tunnettua algoritmin abstraktia mallia, esimerkiksi Turingin konetta. Algoritmi muunnetaan Turingin koneen tila-siirtymärelaatioiksi ja algoritmin toiminnan aiheuttamien tilasiirtymien lukumäärät lasketaan. Tämä ratkaisu ei kuitenkaan ole käytännössä mahdollinen monimutkaisuutensa vuoksi.

Jos tarkastellaan joukkoa algoritmeja, jotka ovat riittävän samankaltaisia, mahdollinen lähestymistapa on etsiä algoritmien suhteellisia eroja jonkin *vertailu-kriteerin* suhteen, ts. kuvataan algoritmit yhtenäisellä tavalla ja verrataan algoritmien suorituksen kannalta oleellisten toimintojen lukumääriä. Tunnettu esimerkki tästä ovat lajittelualgoritmit, jolloin idealisoitu koetilanne käsittää taulukon, joka alussa sisältää satunnaisessa järjestyksessä numerot 1...n; vertailukriteereinä ovat taulukkoalkioiden suuruusvertailujen lukumäärä sekä taulukkoalkioiden paikanvaihtojen lukumäärä. Useimpien lajittelumenetelmien kohdalla vertailukriteerien tarkat arvot n:n funktiona tunnetaan keskimääräisessä ja pahimmassa tapauksessa [ks. esim. Gonnet and Baeza-Yates, 1991].

Mikäli vertailukriteereiden arvoja ei kyetä ratkaisemaan analyyttisesti, ne voidaan selvittää simuloimalla: toteutetaan algoritmi, järjestetään koesarja ja lasketaan tutkittavien suureiden arvot ohjelmallisesti.

Yleinen algoritmien analysointimenetelmä on vertailla kahden tai useamman algoritmin toteutuksen kuluttamaa keskusyksikköaikaa (CPU-aikaa). Tässä tapauksessa menetelmän *ad hoc* -luonne on edellisiä selvempi: tulokset ovat päteviä vain omassa kontekstissaan, jonka parametreja ovat ainakin käytetty laitteistokonfiguraatio ja konekäskykanta sekä ohjelmointikielen kääntäjä. Hyvänä puolena on se, että algoritmin (itse asiassa sen toteutuksen) piilotetut kustannukset sekä operaatioiden todelliset kustannukset tulevat huomioiduiksi.

Seuraavassa rajaamme tarkastelun tässä tutkimuksessa analysoitaviin satunnaisia binääripuita generoiviin algoritmeihin. Ensin esitämme algoritmien analysoinnissa käytettävän Θ-merkintätavan määritelmän soveltuvassa muodossa [ks. esim. Weiss, 1994, s. 32]. Ei-negatiivinen funktio f kuvaa resurssivaatimusta ja c on vakio.

Määritelmä 7.1. Tarkastellaan funktioita $f:N\rightarrow R_+$ ja $g:N\rightarrow R_+$. Jos $\lim_{n\rightarrow\infty} f(n)/g(n) = c$, niin funktio f(n) on $\Theta(g(n))$.

Koska tähän tutkimukseen valittujen algoritmien aikavaatimusten (yhtä lukuunottamatta) tiedetään olevan O(n), pätee g(n)=n. Oletamme, että algoritmien vertailukriteerien suhteen resurssivaatimukset ovat lisäksi $\Theta(n)$.

Tässä tutkimuksessa pyrimme löytämään algoritmeille keskenään vertailukelpoiset vakion c arvot kulloinkin tarkasteltavana olevan kriteerin suhteen. Eri kriteerejä kokonaisuutena tarkastelemalla muodostamme kuvan algoritmien keskinäisistä tehokkuussuhteista. Kutakin vertailukriteeriä erikseen tarkasteltaessa saadaan 'läpileikkaus' algoritmin toiminnasta, jota voidaan verrata muiden algoritmien vastaaviin lukuihin. Kuten edellä todettiin, vakion c arvot ovat suhteellisia, koska ne riippuvat kriteerien valinnasta sekä siitä, missä muodossa algoritmit esitetään.

7.1.2. Toteutus

Tämän tutkimuksen algoritmit generoivat kokonaislukutaulukkoon sijoitettuja binääripuun koodauksia. Koodausten generointi tapahtuu laskemalla todennäköisyyksiä ja satunnaislukujen avulla valitsemalla toteutuvat vaihtoehdot. Laskemme siis käytettyjen satunnaislukujen, aritmeettisten operaatioiden sekä taulukkoviittausten lukumäärät. Yleisempää informaatiota ohjelmien toiminnasta antavat suoritettujen muuttujasijoitusten, muuttujaviittausten, loogisten vertailujen, osoitinviittausten ja aliohjelmakutsujen lukumäärät samoin kuin kustannusten hajonta, muuttujien tarvitsema tila, käytettyjen lukujen suuruus ja keskusyksikköajan kulutus.

Syöttö- tai tulostustoimintojen kustannuksia ei lasketa. Koska satunnaislukugeneraattorin kutsut lasketaan erikseen, niitä ei sisällytetä aliohjelmakutsujen lukumäärään. Lopullista muotoa olevaa binääripuuta rakennettaessa kutsutaan toistuvasti aliohjelmaa, joka varaa dynaamisesti muistitilan lisättävälle puun solmulle. Näitä kutsuja ei lasketa, koska niiden lukumäärä on aina sama kuin puun solmujen lukumäärä. Huomioimatta jätetään myös sellaiset aliohjelmakutsut, joiden ainoa tarkoitus on selkeyttää ohjelman toimintaa ja joiden sisältämä koodi voitaisiin liittää kutsuvan ohjelman osaksi: tällöin lasketaan vain aliohjelman suoritusosan kustannukset, mutta ei itse kutsua (Arnoldin ja Sleepin [1980] algoritmin toteutus). Informaatiota

keräävien ohjelmanosien kustannuksia ei luonnollisestikaan lasketa (operaatioiden lukumääriä käsittelevät ohjelmamodulit[1], tilastollista informaatiota käsittelevät modulit, puun järjestysnumeron evaluoiva moduli).

Ohjelmat sisältävät runsaasti silmukkarakenteita. Silmukoiden ohjausmuuttujien kustannukset lasketaan mukaan sellaisenaan. Esimerkiksi taulukon alkioiden arvojen alustus edellyttää yhden sijoituslauseen suorittamista yhtä taulukon alkiota kohti. Kuitenkin tämän lisäksi silmukan ohjaaminen vaatii jokaista taulukon alkiota kohti mm. yhteen- tai vähennyslaskun sekä loogisen lausekkeen evaluoinnin ja testin. Useissa tapauksissa silmukan kierrosten lukumäärä on tiedossa ennen ohjelman suoritusta ja se on riippumaton mahdollisesta ohjelmalle annettavasta syötteestä (oletamme, että solmujen lukumäärä n on vakio, ei syöte). Tällöin olisi periaatteessa mahdollista purkaa silmukka ja kirjoittaa silmukan sisäiset suoritettavat lauseet peräkkäin, jolloin säästetään silmukan ohjauskustannuksissa ohjelman koon kustannuksella. Tällaisen teoreettisen ‘säästön’ määrä vaihtelee huomattavasti eri algoritmeilla, ja sen avulla ajattelemme saatavan lisätietoa ohjelman silmukoinnin ‘yksinkertaisuudesta’. Sen vuoksi lukujen, jotka sisältävät kaikki kustannukset, lisäksi ilmoitetaan ne kustannukset, jotka saadaan, kun näiden *periaatteessa tarpeettomien* silmukoiden ohjauskustannukset on karsittu.

Seuraavassa on yhteenveto vertailukriteereinä käytettävistä operaatioista. Suluissa ovat niitä vastaavat lyhenteet, joita käytetään taulukoissa.

- aritmeettisten operaatioiden lukumäärä (ADD, MUL, DIV)
- taulukkoviittausten lukumäärä (ARR)
- käytettyjen satunnaislukujen lukumäärä (RAN)
- muuttujaviittausten ja sijoituslauseiden lukumäärä (LOA, STO)
- aritmeettisten ja loogisten vertailujen lukumäärä (CMP)
- rekursiivisten funktiokutsujen lukumäärä (RFUN)
- osoitinmuuttujaviittausten lukumäärä (PTR)
- muita operaatioita, jotka sijoitetaan yhteen kategoriaan (OTH).

[1] *Moduli* vai *moduuli*? Nykysuomen sanakirjassa on *moduuli*. Tämäntyyppisissä vierassanoissa viimeistä edellisen tavun vokaali on useimmiten pitkä, ja se vastaa sanan ruotsinkielisen vastineen (tässä: *modul* [mo'dy:l]) painollista vokaalia [Itkonen, 1988]. Vrt. esim. *banaali*, *modeemi*, *paneeli*, *vinyyli*. Tästä säännöstä on olemassa poikkeuksia, esim. *evankeli(nen)*, *kameli*, *kaneli*, *voguli* (sana *moguli* ei sijoitu kumpaankaan ryhmään). Luontevampaa muotoa *moduli* käytetään tällä hetkellä paljon: tämän voi todeta esim. haulla suomenkielisistä internet-sivuista. Microsoftin, Altavistan ja Sonera Plazan hakupalvelut löysivät kokeiltaessa enemmän sivuja hakusanalla *moduli* kuin *moduuli*. Uudessa Sivistyssanakirjassa [1992] on hakusana *moduli*.

Seuraavassa esitämme esimerkin kustannusten laskennasta. Oletetaan, että lausekielisen ohjelman suorituksen edetessä tavataan lause

A[I] := A[I]+(b+2)/(c-d).

Operaatioiden laskenta suoritetaan täysin mekaanisesti, ilman optimointia, seuraavasti: lause sisältää 8 muuttujan nimeä tai vakiota. Näistä vasemmanpuoleisin on muistipaikkaan sijoitus (STO), ja loput 7 muistipaikkaan viittauksia (LOA). Taulukkoviittauksia (ARR) on kaksi. Yhteen- tai vähennyslaskuja (ADD) on kolme, ja jakolaskuja (DIV) yksi. Kokonais- ja liukulukumuunnokset jätetään huomiotta.

Aritmeettiset ja loogiset vertailulausekkeet evaluoidaan aina loppuun asti, vaikka tulos olisikin tiedossa jo sitä ennen. Täten lauseke (x<y or x<z) sisältää neljä muuttujaviittausta (LOA) ja kolme vertailua (CMP). Toistolause

for i := 1 **to** n **do** **begin** lauseita **end**

tarkoittaa samaa kuin

i := 1
while i≤n **do** **begin** lauseita; i := i + 1 **end** ,

ja sitä vastaava kustannuslauseke, joka sisältää pelkästään silmukan ohjauskustannukset, on (4n+3)*LOA + (n+1)*STO + (n+1)*CMP + n*ADD.

Laskentatapa siis perustuu lausekielisen ohjelman suorituksen seuraamiseen, eikä esim. minkään olemassaolevan tietokoneen suorittamien konekäskyjen laskentaan. Eräitä yhtymäkohtia tosin voidaan todeta ns. abstraktiin pinokoneeseen (abstract stack machine [esim. Aho *et al.*, 1986]), joka on yksinkertaistettu hypoteettinen tietokone, jonka kaikki laskenta tapahtuu ns. laskentapinossa. Esimerkkilauseen a:=b*(c+d) likimääräinen käännös ko. tietokoneen konekielelle on LOA b; LOA c; LOA d; ADD; MUL; STO a. Operandeja siirretään laskentapinoon, ja pinon kahteen ylimmäiseen alkioon kohdistetaan aritmeettinen operaatio; tulos korvaa em. kaksi operandia pinon pinnalla. Pinoa käytettäessä ei aritmeettisen lausekkeen evaluoinnin välituloksia tarvitse varastoida apumuuttujiin, joten operaatioiden ja muuttujaviittausten lukumäärä on sama kuin lausekielisessä esitystavassa.

Kriteereiksi valitut algoritmien perusoperaatiot ovat siis paljolti samat kuin algoritmien ohjelmointikielisten kuvausten operaatiot. On kuitenkin huomattava, että kääntäjän tuottama todellisen tietokoneen konekielinen ohjelmakoodi sisältää lukuisasti erilaisia korkean tason ohjelmointikielessä näkymättömiä operaatioita: esimerkiksi taulukkoviittaus on epäsuora muistiviittaus, mutta konekielinen koodi yleensä sisältää

erilaisia epäsuoria muistiviittauksia myös esim. viittausparametreja (call by reference) käytettäessä, tietueen kenttään viitattaessa tai osoitinmuuttujia käytettäessä [ks. esim. Aho *et al*. 1986].

Kustannusten laskennassa sovellamme yksikkökustannusperiaatetta [ks. esim. Aho and Ullman, 1995, s. 109]. Toisin sanoen yksi operaatio (esim. yhteenlasku, muuttuja-sijoitus) suoritetaan vakioajassa riippumatta käsiteltyjen lukujen suuruudesta. Erilaisilla operaatioilla nämä vakioajat voivat olla erilaiset. Yksikkökustannusperiaatteen käyttöä voidaan pitää hyväksyttävänä, sillä käytetyssä ohjelmointiympäristössä kokonaisluku-tyyppisen muuttujan suurin mahdollinen arvo on rajattu (2^{31}-1) eikä tutkittavissa algoritmeissa jouduta käsittelemään eksponentiaalisia lukuja (poikkeuksena tästä on Johnsenin [1991] algoritmi; suurimman mahdollisen Johnsenin algoritmilla generoitavan puun solmulukumäärä jää tästä syystä alle 20:n).

Tämän tutkimuksen testit keskusyksikköajan mittauksia lukuunottamatta ovat identtisesti toistettavissa antamalla ohjelman suorituksen alussa satunnaislukugeneraattorille samat siemenluvut uudelleen.

Kustannusten laskentaa varten on laadittu seuraavat ohjelmat:

- moduli, jonka sisäinen taulukko sisältää laskettavien operaatioiden frekvenssit
- moduli, joka vaihtaa frekvenssilaskennan ‘kontekstin’; esimerkiksi puiden koodausten generoinnin kustannukset lasketaan eri kontekstissa kuin lopullisten puiden generoinnin kustannukset; eri konteksteissa toteutuneet luvut voidaan lopussa tulostaa erikseen
- moduli, jonka avulla lisätään haluttua frekvenssilaskuria voimassaolevassa kontekstissa; kutsuesimerkki __t(add,2); __t(mul,1) kuvaa tilannetta, jossa on suoritettu kaksi yhteenlaskua ja yksi kertolasku; __t on aliohjelman nimi ja ‘add’ ja ‘mul’ symbolisia vakioita; modulin __t kutsuja lisätään asianmukaisiin kohtiin tutkittavassa ohjelmassa
- modulit, joiden avulla lasketaan painotetut ja painottamattomat kustannukset, keskiarvot ja hajonnat sekä tulostetaan tulokset.

7.2. Toteutusten oikeellisuuden todentaminen tilastollisin menetelmin

Tässä kohdassa esitellään periaatteet, joiden mukaan generointialgoritmien toteutusten oikeellisuutta on tutkittu. Luonnollinen lähestymistapa on tutkia tilastollisesti, ovatko generoidut binääripuut satunnaisia ja noudattavatko ne tasajakaumaa. Jos toteutukset ovat virheettömiä, erilaisten puiden lukumäärät ovat keskimäärin 'riittävän' lähellä odotettuja lukumääriä, mutta eivät *liian* lähellä. Esimerkiksi jos ohjelma tuottaisi täsmälleen saman (suuren) lukumäärän kaikkia erilaisia puita, on todennäköistä, että se on virheellinen: puut noudattavat kyllä tasajakaumaa, mutta eivät ole satunnaisia.

7.2.1. Teoria

Satunnaisen binääripuun generointia voidaan pitää satunnaisilmiönä, jonka tapausavaruus sisältää kaikki n-solmuiset binääripuut. Satunnaisilmiötä kuvaavan satunnaismuuttujan arvo yksilöi satunnaiskokeen tuloksen. Tällaiseksi yksilöiväksi ominaisuudeksi valitsemme binääripuun järjestysnumeron Knottin [1977] järjestyksen perusteella, joten satunnaismuuttuja voi saada arvoja väliltä $1...C_n$. Näin kysymys 'generoiko proseduuri satunnaisia binääripuita siten, että jokainen erilainen puu on yhtä todennäköinen?' voidaan palauttaa helpommin hallittavissa olevaan muotoon 'ovatko puiden perusteella lasketut järjestysnumerot satunnaisia noudattaen tasajakaumaa väliltä $1...C_n$?'.

Kysymyksen ratkaisemiseksi riittävällä tarkkuudella on olemassa monia mahdollisuuksia erilaisten testien muodossa [esim. Knuth, 1969]. Tarkoituksiimme hyvin sopiva testi on χ^2*-yhteensopivuustesti* eli *khiin neliö* -yhteensopivuustesti (Chi-square test for goodness of fit). Se vastaa seuraavantyyppiseen ongelmaan:

> On saatu satunnaisotos muuttujasta, jolla on C_n eri mahdollista arvoa ja saatu arvojen frekvensseiksi $f_1,f_2,...,f_{Cn}$. Kuinka hyvin havaitut frekvenssit ovat sopusoinnussa frekvenssien odotusarvojen kanssa?

Tässä tapauksessa satunnaisotos on siis generoitujen binääripuiden perusteella laskettujen järjestysnumeroiden joukko. Jokaisen havaintoluokan frekvenssin odotusarvo on $(f_1+f_2+...+f_{Cn})/C_n$, jota merkitsemme kirjaimella e.

χ^2-testiä voidaan soveltaa, jos 1) havainnot ovat riippumattomia, 2) havaintojen teoreettinen jakauma eli frekvenssien odotusarvot tiedetään ja 3) havaintoja on tarpeeksi (kukin $e_i \geq 5$). Tarkasteltavan ominaisuuden mittaustasoksi riittää nominaaliasteikko. Formuloidaan nollahypoteesi:

H_0 : Generoitujen puiden järjestysnumerot ovat satunnaisia.

Mittaamme lukujen satunnaisuutta kaksisuuntaisella χ^2-testillä [Knuth, 1969, s. 35]. Mikäli H_0 on tosi, noudattaa testisuure

$$X^2 = \sum_{i=1}^{Cn} (f_i\text{-}e)^2 / e$$

likimäärin χ^2-jakaumaa C_n-1:llä vapausasteella (degrees of freedom).

Jos havaitut frekvenssit poikkeavat paljon odotetuista, testisuureen X^2 arvo on suuri; päinvastaisessa tapauksessa X^2:n arvo on lähellä nollaa. Kummassakin 'ääritapauksessa' havaintojen satunnaisuutta voidaan epäillä, varsinkin jos ilmiö ei ole kertaluontoinen. Koska kokeemme ovat uusittavissa, emme katso tarpeelliseksi kiinnittää mitään erityistä merkitsevyystasoa, jonka ylittävillä (vast. alittavilla) X^2:n arvoilla automaattisesti hylkäämme H_0:n. Sen sijaan ilmoitamme todennäköisyyden sille, että H_0:n voimassaollessa saadaan havaittu tai sitä suurempi (vast. havaittu tai sitä pienempi) X^2:n arvo, ja teemme johtopäätöksiä tapauskohtaisesti.

Pienillä vapausasteiden lukumäärillä χ^2-jakauman arvot vapausasteittain ja merkitsevyystasoittain löytyvät tilastollista taulukoista. Jos vapausasteiden lukumäärä df on suuri, χ^2:n likimääräiset arvot voidaan selvittää mm. seuraavan muunnoksen avulla [ks. esim. Clarke and Cooke, 1988]:

(7.2) $$\sqrt{2\,c^2} \sim \text{Norm}(\sqrt{2\,\text{df} - 1}, 1), \qquad \text{df} > 100,$$

missä merkinnällä Norm(a,b) tarkoitamme satunnaismuuttujan arvoja, jotka noudattavat normaalijakaumaa siten, että niiden keskiarvo on a ja keskihajonta b. Sadan vapausasteen raja ylittyy jo kuusisolmuisten puiden tapauksessa (C_6=132, df=131).

Lopuksi esitämme esimerkkejä χ^2:n arvojen käytöstä. Olettakaamme, että generoimme nelisolmuisia binääripuita. Vapausasteiden lukumäärä on 13, koska C_n=14. χ^2-jakauman taulukoitu arvo $\chi^2_{0.9;13}$=19.8 tulkitaan seuraavasti: jos H_0 on voimassa, niin satunnaisotoksesta laskettu X^2:n arvo on todennäköisyydellä 0.9 pienempi kuin 19.8. Toisaalta, koska $\chi^2_{0.1;13}$=7.04, niin keskimäärin 10 prosenttia satunnaisotosten perusteella lasketuista X^2:n arvoista on tätä lukua pienempiä ja 90 prosenttia sitä

suurempia. Mediaaniarvo on $\chi^2_{0.5;13}$=12.3.

Toisessa esimerkissä generoimme kymmensolmuisia binääripuita (df=16795), ja testisuureen X^2 arvoksi olemme saaneet 17000. Kaavan 7.2 vasemman puolen mukainen testisuureen arvo on $\sqrt{2*17000}$ =184.39 ja oikean puolen kuvaama jakauma on Norm(183.27, 1). Vastaava normaalijakauman standardoitu havaintoarvo on z = (184.39-183.27)/1 = 1.12. H_0:n voimassaollessa todennäköisyys sille, että z <1.12, on 0.87.

7.2.2. Toteutus

Khiin neliön arvon laskemiseksi satunnaisia binääripuita generoivien toteutusten perusteella on laadittu seuraavat ohjelmat:

- binääripuun solmujen läpikäynti ja numerointi symmetrisessä järjestyksessä luvuilla 1...n
- puupermutaation muodostaminen edellämainitun numeroinnin perusteella; puun solmut siis läpikäydään ja listataan esijärjestyksessä
- puun järjestysnumeron laskenta puupermutaation perusteella: tämä on Knottin [1977] kokonaislukufunktio *rank*
- tilastoaineiston eli järjestysnumerojen frekvenssien talletus
- X^2:n arvon laskenta tilastoaineiston perusteella.

7.3. Satunnaislukugeneraattori

Satunnaislukujen käyttö on olennainen osa satunnaisia binääripuita generoivien algoritmien toimintaa. Koska tässä tutkimuksessa esitetyt toteutukset testataan tilastollisesti, on välttämätöntä, että käytetyn (pseudo)satunnaislukugeneraattorin tuottamat luvut täyttävät (pseudo)satunnaisluvuille asetettavat vähimmäisvaatimukset. Tässä kohdassa tarkastelemme käytettyyn satunnaislukugeneraattoriin liittyviä kysymyksiä.

7.3.1. Teoria

Satunnaisten lukujen generointi sekä lukujonon ja -joukon satunnaisuuden tutkiminen muodostavat oman tieteenalansa, joka sisältää elementtejä tilastotieteestä ja lukuteoriasta. Tämän tutkimuksen toteutuksissa käytetään satunnaislukugeneraattoria,

jonka ominaisuudet tunnetaan hyvin ja joka läpäisee tunnetut tilastolliset testit riittävän hyvällä menestyksellä. Ko. generaattori käyttää Lehmerin v. 1951 keksimää lineaarista kongruenssimenetelmää (linear congruential method, ks. esim. Knuth [1969]), jonka yleinen muoto on

$$z_{i+1} = az_i \mod b, \tag{7.3}$$

missä a ja b ovat kokonaislukuvakioita ja z_i on kokonaislukumuuttuja, jolle generoinnin alussa asetetaan mielivaltainen alkuarvo eli *siemenluku* (seed) väliltä $1 \leq z_0 < b$. Lukujen a ja b valinta ratkaisee, millaiset generaattorin ominaisuudet ovat. Satunnaisluku-generaattorissamme on $a=7^5$ ja $b=2^{31}-1$, jolloin se tuottaa $b-1>2*10^9$ kokonaislukua ilman toistoja näennäisen satunnaisessa järjestyksessä, jonka jälkeen jono alkaa alusta. Valittu luku b on suurin mahdollinen silloin, kun tietokoneen sananpituus on 32 bittiä ja kokonaislukujen sisäinen esitysmuoto on 2:n komplementtijärjestelmä. Tätä satunnais-lukugeneraattoria Park and Miller [1988] ehdottavat ‘minimistandardiksi’, johon muita generaattoreita voidaan verrata.

Kun siemenluku on annettu, satunnaislukugeneraattorin tuottama lukujono on tietenkin täysin deterministinen ja toistettavissa. Lukujonoa ei kuitenkaan tilastollisin menetelmin tutkimalla voida erottaa sellaisesta lukujonosta, joka saadaan, kun poimitaan ‘aidon’ satunnaisesti lukuja lukujoukosta {1,2,...,b-1} ilman palautusta [Park and Miller, 1988]. Tätä kuvaa nimitys ‘pseudosatunnaisluku’.

Satunnaislukugeneraattorin tuottamat luvut z_i jaetaan tavallisesti luvulla b, jolloin satunnaisluvut $u_i=z_i/b$ ovat rationaalilukuja väliltä $0<u_i<1$. Huomaa, että $z_i=u_i=0$ on mahdoton, koska b on alkuluku. Tarkasti ottaen tarkoituksiimme soveltuvampi olisi väli $0 \leq u_i < 1$ (joka on saavutettavissa jakolaskulla $u_i=z_i/(b-1)$), mutta pitäydymme tavan-omaisessa menettelyssä, koska ero on käytännössä merkityksetön.

7.3.2. Toteutus

Tässä tutkimuksessa käytetyn minimistandardigeneraattorin toteutus on Parkin ja Millerin [1988] artikkelissa esitetty “Integer version 2”.

Kaava 7.3 ei sellaisenaan sovellu ohjelmointikielellä toteutettavaksi ympäristössä, jossa tietokoneen sananpituus on 32 bittiä, koska kertolasku az_i aiheuttaa ennemmin tai myöhemmin ylivuodon. Esimerkkiohjelmia, joissa tämä ongelma on ratkaistu, esittävät mm. Park ja Miller [1988] (mm. “Integer version 2”), Carta [1990] ja Weiss [1994].

Park ja Miller antavat myös yksinkertaisen tavan testata ja tarkistaa minimi-standardigeneraattorin toteutus: jos z_1=1, niin z_{10001}=1 043 618 065.

7.4. Binääripuun konstruointialgoritmit koodausten perusteella

Tässä kohdassa esitellään algoritmit, jotka konstruoivat lopullisen binääripuun, kun syötteenä on binääripuun koodaus. Resurssivaatimuksia tutkitaan sekä algoritmien generoimien koodausten että lopullista muotoa olevien binääripuiden perusteella, koska näin varmistutaan eri algoritmien tekemien työmäärien yhteismitallisuudesta. Sivutuloksena saadaan selville, vaihteleeko binääripuun konstruoinnin kustannus koodauksen perusteella merkittävästi eri koodausmenetelmissä.

Binääripuiden konstruointialgoritmit esitetään pseudokielisinä rekursiivisina proseduureina. Kaikissa tapauksissa puu rakennetaan esijärjestyksessä, ja rekursiivisten proseduurikutsujen muodostaman implisiittisen puun rakenne on sama kuin generoitavan puun rakenne. Uuden solmun luontioperaatio palauttaa arvonaan dynaamisesti varatun tietoalkion osoitteen; lisäksi solmun osoitinkentät vasempaan ja oikeaan lapseen alustetaan tyhjillä arvoilla, jota merkitsemme varatulla sanalla NIL. Parametrilistoissa mainitut muuttujat ovat viittausparametreja (call by reference), ts. aliohjelma voi muuttaa niiden arvoa. Proseduurit olettavat, että syötteenä annetut koodaukset ovat ei-tyhjiä ja virheettömiä.

7.4.1. Tasapainotetut sulkujonot

Algoritmi 7.4 rakentaa binääripuun tasapainotetun sulkujonon [Arnold and Sleep, 1980; Atkinson and Sack, 1992] perusteella. Kutsu on muotoa root := NIL; treeBuild_Parentheses(root), missä root on puun juuren osoitin.

```
procedure treeBuild_Parentheses(solmu)
x := lue seuraava syötemerkki
if  (x = alkusulku) then
   solmu := luo uusi solmu
   treeBuild_Parentheses(solmun vasen lapsi)
   if  (syötemerkkejä jäljellä)
       then  treeBuild_Parentheses(solmun oikea lapsi)  end if
end if
end procedure
```

Algoritmi 7.4. Binääripuun luonti tasapainotetun sulkujonon perusteella.

7.4.2. Oikean etäisyyden koodisanat

Algoritmi 7.5 rakentaa binääripuun Martinin ja Orrin [1990] koodausmenetelmän (käännöstaulu-/oikean etäisyyden menetelmä) perusteella. Kutsu on muotoa root := NIL; x := 0; treeBuild_RightDistance(root, x).

```
procedure treeBuild_RightDistance(solmu, x)
integer  cd
solmu := luo uusi solmu
if  (kaikki syötemerkit luettu)
then  x := -1
else
   cd := x;    x := lue seuraava syötemerkki
   if  (x > cd)
       then  treeBuild_RightDistance(solmun vasen lapsi, x)  end if
   if  (x = cd)
       then  treeBuild_RightDistance(solmun oikea lapsi, x)  end if
end if
end procedure
```

Algoritmi 7.5. Binääripuun luonti oikean etäisyyden koodausmenetelmän perusteella.

7.4.3. Oksastus

Algoritmi 7.6 rakentaa binääripuun Johnsenin [1991] koodausmenetelmän perusteella. Kutsu on muotoa root := NIL; x := 0; treeBuild_Grafting(root, x). Vaihtoehtoisen tavan konstruoida puu esitti Mäkinen [1991b].

```
procedure treeBuild_Grafting(solmu, x)
solmu := luo uusi solmu
if  (kaikki syötemerkit luettu)
then  x := -1
else
   x := lue seuraava syötemerkki
   if  (x = 0)
       then  treeBuild_Grafting(solmun vasen lapsi, x)  end if
   if  (x = 1)
       then  treeBuild_Grafting(solmun oikea lapsi, x)
       else   x := x - 1 end if
end if
end procedure
```

Algoritmi 7.6. Binääripuun luonti Johnsenin [1991] koodausmenetelmän perusteella.

7.4.4. Esijärjestys-numeroparit

Algoritmi 7.7 rakentaa binääripuun Korshin [1993] koodausmenetelmän perusteella. Kutsu on muotoa root := NIL; treeBuild_PreOrderPair(root). Ohjelmassa ei tarvita lainkaan syötemerkkien loppumisen tutkintaa, koska tämä tieto sisältyy koodausmenetelmään.

```
procedure treeBuild_PreOrderPair(solmu)
integer left, right
left := lue seuraava syötemerkki;  right := lue seuraava syötemerkki
solmu := luo uusi solmu
if  (left   = 1)
     then  treeBuild_PreOrderPair(solmun vasen lapsi) end if
if  (right = 1)
     then  treeBuild_PreOrderPair(solmun oikea lapsi) end if
end procedure
```

Algoritmi 7.7. Binääripuun luonti Korshin [1993] koodausmenetelmän perusteella.

7.4.5. Solmujen taulukointi

Algoritmi 7.8 rakentaa binääripuun Alonson ja Schottin [1995] koodausmenetelmän perusteella. Puun solmut on koodattu globaaliin taulukkoon treeArr, jonka kukin alkio on tietue, jonka kentät ovat vanhempi, vasenLapsi ja oikeaLapsi, jotka puolestaan ovat indeksejä toisaalle taulukkoon. Lehtisolmut sisältyvät taulukkoon. Lehtisolmujen lasten sekä juuren vanhemman arvot ovat NIL. Kutsu on muotoa root := NIL; ind := 0; treeBuild_Array(root, ind).

Vaikka algoritmi 7.8 saattaakin näyttää yksinkertaisemmalta kuin muut tämän luvun algoritmit, on huomattava, että se ‘olettaa’ koodauksen sisältävältä tietorakenteelta enemmän kuin muut koodausmenetelmät: se on suorasaantirakenne, kun taas muiden menetelmien koodaukset ovat peräkkäisrakenteita.

```
procedure treeBuild_Array(solmu, ind)
if  (treeArr(ind).vasenLapsi ≠ NIL  or  treeArr(ind).oikeaLapsi ≠ NIL) then
    solmu := luo uusi solmu
     treeBuild_Array(solmun vasen lapsi, treeArr(ind).vasenLapsi)
     treeBuild_Array(solmun oikea lapsi, treeArr(ind).oikeaLapsi)
end if
end procedure
```

Algoritmi 7.8. Binääripuun luonti Alonson ja Schottin [1995] koodausmenetelmän perusteella.

7.5. Rémyn algoritmin toteutus

Yksi tässä tutkimuksessa tarkasteltavista algoritmeista on Rémyn satunnaisia binääripuita generoiva algoritmi Alonson ja Schottin [1995, Appendix 2, Alg. 2.1] esimerkkitoteutuksen mukaisesti. Ko. esimerkkitoteutus kuitenkin on virheellinen: sen generoimat binääripuut ovat kylläkin ‘satunnaisia’, mutta eivät noudata tasajakaumaa. Virhe löydettiin khiin neliö -testin avulla. Seuraavassa tutkimme virheellisen toteutuksen toimintaa. Tämän kohdan tulokset on esitetty myös erillisessä raportissa [Mäkinen and Siltaneva, 2000].

Alonson ja Schottin toteutuksen soveltamassa binääripuiden koodauksessa puun solmut talletetaan taulukkoon tietueina (ks. myös kohta 7.4.5). Tietueen kentät ovat kokonaislukumuuttujat vanhempi, vasenLapsi ja oikeaLapsi, jotka puolestaan ovat indeksejä toisaalle taulukkoon. Puun juuren tiedot ovat taulukon positiossa 0. Lehtisolmut sisältyvät taulukkoon. Lehtisolmujen lasten sekä juuren vanhemman arvot ovat NIL.

Ohjelma rakentaa puun seuraavasti. Puun k sisäsolmua ovat aina taulukon positioissa 0...k-1 ja lehtisolmut positioissa k...2k. Taulukon positio j arvotaan väliltä k...2k. Positioiden k ja j (siis k=j on myös mahdollinen) tiedot ‘vaihtavat paikkaa’ (proseduuri *change_leaves*). Tällöin puun muoto ei muutu, ainoastaan lehtisolmujen taulukkopositiot. Tämän jälkeen pienimmässä positiossa (=k) oleva lehtisolmu muutetaan sisäsolmuksi ja taulukon loppuun lisätään kaksi lehtisolmua sen lapsiksi. Näin jatketaan kunnes puussa on n sisäsolmua.

Seuraavassa esitämme pienimmän mahdollisen (n=3) vastaesimerkin, jolloin toteutuksen generoimat puut eivät noudata tasajakaumaa. Tämä vaatii tyhjentävän kartoituksen ohjelman toiminnasta mainitussa tapauksessa. Havainnollisuussyistä olemme pakotettuja käyttämään ‘koodauksen koodausta’, eli esitämme Alonson ja Schottin käyttämän koodauksen toisessa muodossa. Kartoitus on esitetty kuvassa 7.9, jossa käytetty puiden esitystapa on seuraava: 1) numero tarkoittaa puun solmua, jonka tiedot on talletettu numeron ilmaisemaan taulukon positioon; 2) sisäsolmun perässä sulkujen sisällä ovat ensin vasemman ja sen jälkeen oikean lapsen tiedot tätä merkintätapaa käyttäen esitettyinä; 3) jos numeroa ei seuraa alkusulku, solmu on lehtisolmu.

Käytettyä koodausmenetelmää selventää seuraava esimerkki: 0(2 1(4 3)) esittää Alonson ja Schottin koodausta, jossa binääripuun juurisolmun tiedot on talletettu taulukon positioon 0; juuren vasen lapsi on lehtisolmu, jonka tiedot ovat taulukon

positiossa 2; juuren oikea lapsi on sisäsolmu, jonka tiedot ovat taulukon positiossa 1, ja tämän lapset ovat kaksi lehtisolmua positioissa 4 ja 3.

```
(1)              0(1 2)
(2)                 0(1 2) → 0(1(4 3) 2)
(3)                                0(1(4 3) 2) → 0(1(4 3) 2(6 5))
(4)                                0(1(4 2) 3) → 0(1(4 2(6 5)) 3)
(5)                                0(1(2 3) 4) → 0(1(2(6 5) 3) 4)
(6)                 0(2 1) → 0(2 1(4 3))
(7)                                0(2 1(4 3)) → 0(2(6 5) 1(4 3))
(8)                                0(3 1(4 2)) → 0(3 1(4 2(6 5)))
(9)                                0(4 1(2 3)) → 0(4 1(2(6 5) 3))
```

Kuva 7.9. Kaikki mahdolliset Alonson ja Schottin [1995, Appendix 2, Alg. 2.1] ohjelman tuottamat binääripuun koodaukset, kun n=3.

Kuvan 7.9 riviltä 1 nähdään, että ohjelman alkutilanne on puu 0(1 2). Satunnaislukugeneraattorin käytön jälkeen jatketaan yhtä suurella todennäköisyydellä joko rivin 2 tilanteesta tai rivin 6 tilanteesta eli joko puusta 0(1 2) tai puusta 0(2 1). Nuoli tarkoittaa sitä, että vasemmanpuoleisesta puusta rakennetaan oikeanpuoleinen puu ilman valintoja. Jos päädyttiin riville 2, jatketaan joko riville 3, 4 tai 5, missä kunkin todennäköisyys on 1/3. Jos taas päädyttiin riville 6, jatketaan joko riville 7, 8 tai 9, missä kunkin todennäköisyys on 1/3.

Lopullisia koodauksia tutkimalla huomaamme, että rivien 3 ja 7 puiden muoto on sama. Ko. rivien koodaukset eroavat siten, että puun solmujen tiedot on sijoitettu eri positioihin taulukossa, mutta molemmat generoivat samanlaisen binääripuun. Koska kaikki mahdolliset 6 koodausta ovat yhtä todennäköisiä, tulemme siihen tulokseen, että tutkittavana olevan ohjelman generoimat binääripuut eivät noudata tasajakaumaa ainakaan kolmisolmuisen puun tapauksessa. Lisäksi khiin neliö -testin perusteella havaittiin, että generoidut puut eivät noudata tasajakaumaa tutkituilla n:n arvoilla ($n \leq 10$). Tällöin khiin neliö -testisuureiden arvot olivat sitä suurempia, mitä pitemmästä koesarjasta oli kyse.

Tätä tutkimusta varten Rémyn satunnaisia binääripuita generoiva algoritmi on toteutettu uudelleen Alonson ja Schottin käyttämää koodausmenetelmää soveltaen. Solmua lisättäessä ohjelma arpoo sekä yhden puun $2n+1$ solmusta että suunnan oikea/vasen. Taulukon loppuun lisätään uusi sisäsolmu ja uusi lehtisolmu. Solmujen vanhempi- ja lapsimuuttujen arvot asetellaan niin, että uuden sisäsolmun oikeaksi/vasemmaksi lapseksi tulee arvottu solmu ja uuden sisäsolmun toiseksi lapseksi uusi lehtisolmu (ks. luku 5.3). Koska puun juuren taulukkopositio saattaa puun konstruoinnin aikana muuttua, ohjelma ylläpitää erityistä puun juuren osoitinta.

8. Tutkimustulokset

Tässä luvussa esitellään testiajojen tulokset. Vertailukriteerien asymptoottisia arvoja tutkittiin vaihtelemalla testisarjojen pituuksia ja puiden solmulukumääriä. Algoritmien toteutusten oikeellisuutta tutkittiin khiin neliö -testillä.

Käytämme algoritmeista kolmimerkkisiä lyhenteitä ARN, MAR, ATK, JOH, KOR ja REM, jotka saadaan algoritmin (aakkosjärjestyksessä ensimmäisen) keksijän kolmesta ensimmäisestä kirjaimesta: Arnold ja Sleep [1980], Martin ja Orr [1990], Atkinson ja Sack [1992], Johnsen [1991], Korsh [1993] sekä Rémyn algoritmi vuodelta 1985 [Mäkinen and Siltaneva, 2000]. Puun solmujen lukumäärää merkitsemme n:llä ja generoitujen puiden lukumäärää r:llä. Taulukoissa käytämme kirjainta K merkitsemään lukua tuhat, esim. 15000=15K.

Algoritmin JOH osalta jätämme tässä luvussa kokonaan huomiotta esitaulukointi-vaiheen kustannukset, jotka ovat kvadraattisia. Lisäksi algoritmin JOH mittaustulosten tulkinnassa on muistettava se, että suurin suorituskykytesteissä käytetty n:n arvo oli 10, kun muiden algoritmien osalta se oli 10^5 tai 10^6.

8.1. Toteutusten oikeellisuus

Generoitujen puiden satunnaisuutta (eli testihypoteesia H_0) testattiin soveltamalla khiin neliö -testiä kohdan 7.2.2 mukaisesti. Tilastollisen luotettavuuden eräs edellytys on generoitavien puiden riittävän suuri lukumäärä ($>5C_n$). Koska erilaisten puiden lukumäärän kasvu solmulukumäärän n funktiona on eksponentiaalista, myös testeissä tarvittava generoitavien puiden lukumäärä kasvaa eksponentiaalisesti, ja tämä asettaa rajoituksensa puiden koolle. Tässä tapauksessa ylärajaksi asetettiin n=10. Sitä suuremmilla n:n arvoilla oletamme toteutusten generoimien puiden olevan satunnaisia, ellei hypoteesia H_0 pystytä kumoamaan.

Taulukossa 8.1 on listattu kyseeseen tulevat khiin neliö -jakauman viitearvot. Merkitsemme df:llä vapausasteiden lukumäärää. Esimerkiksi jos n=4, niin satunnaisotoksen perusteella lasketun testisuureen arvo on 95% todennäköisyydellä <22.36, mikäli H_0 on tosi.

n	df	.01	.05	.10	.25	.50	.75	.90	.95	.99
3	4	.2971	.7107	1.064	1.923	3.357	5.385	7.779	9.487	13.28
4	13	4.107	5.891	7.041	9.299	12.34	15.98	19.81	22.36	27.69
5	41	22.91	27.33	29.91	34.58	40.34	46.69	52.95	56.94	64.95
6	131	96.31	105.6	110.7	119.8	130.3	141.5	152.1	158.7	171.6
7	428	362.9	381	391	407.9	427.3	447.4	465.9	477.2	499
8	1429	1308	1342	1361	1393	1428	1465	1498	1518	1556
9	4861	4635	4700	4735	4794	4860	4927	4988	5024	5093
10	16795	16372	16495	16561	16671	16794	16918	17030	17098	17224

Taulukko 8.1. Khiin neliö -jakauman viitearvoja.

Taulukossa 8.2 ovat suoritetun testisarjan perusteella saadut khiin neliö -testisuureiden arvot.

n	df	r	ARN	MAR	ATK	JOH	KOR	REM
3	4	10K	10.11	0.54	3.84	6.38	2.86	6.36
4	13	10K	5.94	7.94	12.39	12.22	14.77	19.45
5	41	10K	53.62	39.88	40.08	44.34	19.44	37.66
6	131	10K	136.36	126.56	103.91	134.70	142.96	126.27
7	428	10K	405.48	404.71	399.90	394.15	454.56	365.07
8	1429	15K	1498.20	1467.50	1395.81	1482.94	1387.04	1337.08
9	4861	30K	4782.38	4815.12	4623.57	4975.57	4873.80	4822.90
10	16795	100K	16990.04	16447.06	16960.46	16750.23	16988.31	16759.93

Taulukko 8.2. Yhden koeajosarjan havainnoituja khiin neliö -testisuureen arvoja.

Taulukon 8.2 havainnot voidaan yhteismitallistaa muuntamalla ne P-arvoiksi eli todennäköisyyksiksi (taulukko 8.3). Esimerkkinä tarkastelemme algoritmin MAR havaintoa 7.94: kun n=4 ja H_0 on tosi, niin todennäköisyys sille, että mielivaltainen havainnoitu khiin neliön arvo on pienempi kuin 7.94, on 0.1525. Taulukon 8.3 vasemmanpuoleiset luvut on haettu taulukon 8.2 lukujen perusteella elektronisesta tilastollisesta taulukosta [Dear, 1997]. Taulukon 8.3 oikealla puolella ovat uusinta-ajon perusteella saadut luvut. Viimeisellä rivillä ovat sarakkeiden keskiarvot. Jos H_0 on tosi, niin taulukon 8.3 luvut itse asiassa ovat (likimäärin, vrt. kaavan 8.4 huomautus) tasajakaumaa noudattavan reaaliarvoisen satunnaismuuttujan arvoja väliltä [0,1]. Tähän seikkaan perustuen suoritamme seuraavassa todennäköisyyslaskelmia.

n	df	r	ARN	MAR	ATK	JOH	KOR	REM	ARN	MAR	ATK	JOH	KOR	REM
3	4	10K	.9614	.0305	.5719	.8275	.4185	.8262	.7549	.9012	.9361	.6191	.7054	.6430
4	13	10K	.0517	.1525	.5041	.4903	.6781	.8902	.2034	.9104	.3758	.9164	.4018	.9891
5	41	10K	.9105	.4797	.4886	.6673	.0017	.3801	.6871	.6393	.3788	.8824	.9737	.0998
6	131	10K	.6435	.4067	.0389	.6055	.7759	.3996	.9435	.1476	.6684	.9240	.2669	.3494
7	428	10K	.2234	.2153	.1687	.1218	.8192	.0124	.3030	.2031	.7880	.7937	.3706	.2252
8	1429	15K	.9009	.7662	.2699	.8435	.2176	.0405	.6731	.4721	.3810	.2723	.5502	.8517
9	4861	30K	.2133	.3228	.0073	.8769	.5543	.3517	.4159	.1528	.7272	.8125	.6328	.2352
10	16795	100K	.8563	.0282	.8169	.4048	.8541	.4255	.3516	.3621	.2287	.4034	.6628	.3935
			.595	.300	.358	.605	.540	.416	.541	.474	.561	.703	.571	.473

Taulukko 8.3. Kahden koeajosarjan khiin neliö -havainnot P-arvoina.

Rajoitumme tässä kohdassa tutkimaan pelkästään taulukon 8.3 vasemmanpuoleisen testisarjan lukuja. Tutkimme niitä kahden eri näkemyksen kautta: 1) etsimme yksittäisiä äärihavaintoja ja analysoimme niiden esiintymisen todennäköisyyttä sekä 2) analysoimme vastaavalla tavalla erikseen kunkin algoritmin tuottamia lukuja.

1) Kahta lukuunottamatta yksittäiset khiin neliö -testisuureen arvot osuvat taulukon 8.1 rajoihin. Suurin poikkeama havaitaan algoritmin KOR tapauksessa, kun n=5. Tällöin generoitujen puiden jakauma oli sellainen, että sitä 'tasaisemman' jakauman todennäköisyys on noin 0.0017 = 1/588, jos H_0 on tosi.

Kuinka todennäköistä on, että testisarjassa, joka koostui 6*8=48 erillisestä ajosta, esiintyy ainakin yksi niin tasainen generoitujen puiden jakauma, että sitä tasaisemman jakauman todennäköisyys on vain 1/588? Vastaavan kysymyksen itse asiassa kysyisimme, mikäli testisarja sisältäisi niin 'epätasaisen' jakauman, että sitä epätasaisemman jakauman todennäköisyys olisi 1/588. Todennäköisyys p sille, että jompikumpi tai molemmat edellämainituista tapauksista toteutuu, on

$$p = 1 - (1 - 2*0.0017)^{48} = 0.1508 = 1/6.63.$$

Toisin sanoen, jos H_0 on tosi kaikkien algoritmien osalta, tarkasteltua arvoa 'äärimmäisempi' arvo esiintyy taulukon 8.3 kaltaisissa taulukoissa keskimäärin noin joka seitsemännessä.

2) Seuraavaksi tarkastelemme algoritmien koesarjoja erikseen. Taulukossa 8.3 odotusarvosta (noin) 0.5 eniten poikkesi koesarjan MAR lukujen keskiarvo 0.300. Kuinka todennäköistä on, että kahdeksan koeajon joukossa esiintyy keskiarvo, joka on <0.300 (tai vastaavasti >0.700), jos H_0 on tosi? Mallinnamme ongelman seuraavasti: olkoon $S_m = X_1+X_2+...+X_m$, missä kukin X_i on riippumaton satunnaismuuttuja, joka generoi (tarkalleen) tasajakaumaa noudattavia reaalilukuja väliltä [0,1]. S_m:n jakauma lähestyy normaalijakaumaa, kun m kasvaa (keskeinen raja-arvolause). Kuitenkin tässä tapauksessa m on liian pieni, jotta normaalijakauma-approksimaatio olisi riittävä. Avuksi tarvitaan S_m:n kertymäfunktio, jonka antaa Feller [1966, s. 27]. Todennäköisyys sille, että $S_m \le x$, missä x on mielivaltainen reaaliluku, on

$$P\{S_m \le x\} = \frac{1}{m!} \sum_{i=0}^{m} (-1)^i \binom{m}{i} (\mathrm{Pos}(x-i))^m , \tag{8.4}$$

missä funktio Pos(x)=x, kun x≥0, ja Pos(x)=0, kun x<0. Mainittu todennäköisyys on aina 0, kun x≤0, ja aina 1, kun x≥m, ja $P\{S_m \le m/2\}=0.5$. Huomaa kuitenkin, että khiin neliö -testisuureen likimääräisyyden vuoksi (χ^2-jakauman viitearvoihin verrattuna)

P-arvojen summan jakauman ja kaavan 8.4 välinen vastaavuus ei ole tarkka. Testisuureen arvot (jos H_0 on tosi) noudattavat χ^2-jakaumaa sitä tarkemmin, mitä suurempi r on. Tarkkuuteen vaikuttavia tekijöitä ovat lisäksi vapausasteiden lukumäärä sekä havaintoluokkien todennäköisyydet.

Taulukossa 8.3 algoritmin MAR arvojen summa on 2.4019. Kaavan 8.4 perusteella arvioimme $P\{S_8 \leq 2.4019\} = (2.4019^8 - 8*1.4019^8 + 28*0.4019^8) / 8! = 0.0245$. Symmetrian perusteella on $P\{S_8 \leq 8-2.4019\} = 1-0.0245$. Täten todennäköisyys vähintään havaitunsuuruisen poikkeaman (suuntaan tai toiseen) esiintymiseen on noin $2*0.0245 = 0.049$, jos H_0 on tosi. Algoritmeja on kuitenkin kuusi, ja kiinnitimme huomiomme 'poikkeavimpaan' tapaukseen. Mikä on todennäköisyys sille, että H_0:n voimassaollessa kuudesta tapauksesta vähintään yhdessä esiintyy havaitunsuuruinen tai suurempi poikkeama odotusarvosta? Tämä todennäköisyys on $p = 1-(1-2*0.0245)^6 = 0.260$.

Tässä kohdassa esitetyt todennäköisyyslaskelmat ovat esimerkkejä. Vastaavankaltaisia laskelmia voidaan esittää näkökulmasta riippuen monia erilaisia. Testiajoja uusittaessa saadaan tarkentuva kuva tutkittavien lukujen satunnaisuudesta. Uusintaajoissa havaittiin, että 'epäilyttäviä' arvoja esiintyi, mutta ne eivät esiintyneet samoissa kohdissa ajoja toistettaessa. Toisin sanoen, epäilyttävien arvojen esiintyminen oli satunnaista. Toteamme siis, että emme voi hylätä testihypoteesia H_0 suoritettujen khiin neliö -testisarjojen perusteella.

8.2. Koodausten generointi: operaatioiden lukumäärät

Tässä kohdassa taulukoimme vertailukriteereinä käytettyjen operaatioiden havaittuja lukumääriä eri algoritmien toteutuksilla ja eri n:n arvoilla (taulukot 8.5 - 8.10). Menettely on seuraava: kiinnitetään generoitavien puiden koko n ja lukumäärä r, lasketaan yhteen koko testiajon operaatioiden ABC lukumäärä, jaetaan tämä lukumäärä luvulla n*r ja ilmoitetaan tulos sarakkeessa ABC. Näin saadut luvut itse asiassa ilmoittavat keskimääräisen operaatioiden lukumäärän yhtä solmua kohti, ja ne ovat vertailukelpoisia riippumatta testisarjan pituudesta sekä puun koosta, kunhan testisarja on riittävän pitkä. Suurilla n:n arvoilla testisarjan 'riittävästä' pituudesta jouduttiin tinkimään. Puiden kokoa kasvattamalla toivottiin saatavan selville vertailukriteerien asymptoottiset raja-arvot. Operaatioiden nimien selitykset annettiin kohdassa 7.1.2.

n	r	LOA	STO	CMP	ADD	MUL	DIV	ARR	FUN	RFUN	RAN	OTH
4	10K	34.99	10.75	5.00	10.25	2.00	1.00	2.00	0.00	0.00	1.00	0.00
10	10K	39.29	11.30	5.45	12.10	3.00	1.50	2.00	0.00	0.00	1.50	0.00
100	10K	43.44	11.91	5.93	13.78	3.88	1.94	2.00	0.00	0.00	1.94	0.00
1K	5K	43.94	11.99	5.99	13.98	3.99	1.99	2.00	0.00	0.00	1.99	0.00
10K	1K	43.99	12.00	6.00	14.00	4.00	2.00	2.00	0.00	0.00	2.00	0.00
100K	100	44.00	12.00	6.00	14.00	4.00	2.00	2.00	0.00	0.00	2.00	0.00

Taulukko 8.5. Koodausten generointi: operaatioiden lukumäärät [Arnold ja Sleep, 1980].

n	r	LOA	STO	CMP	ADD	MUL	DIV	ARR	FUN	RFUN	RAN	OTH
4	10K	32.61	7.77	2.26	15.06	3.02	1.26	1.75	0.00	0.00	0.75	0.00
10	10K	41.30	9.50	2.65	19.80	4.05	1.65	1.90	0.00	0.00	0.90	0.00
100	10K	48.13	10.83	2.96	23.53	4.89	1.96	1.99	0.00	0.00	0.99	0.00
1K	5K	48.91	10.98	3.00	23.95	4.99	2.00	2.00	0.00	0.00	1.00	0.00
10K	1K	48.99	11.00	3.00	24.00	5.00	2.00	2.00	0.00	0.00	1.00	0.00
100K	100	49.00	11.00	3.00	24.00	5.00	2.00	2.00	0.00	0.00	1.00	0.00

Taulukko 8.6. Koodausten generointi: operaatioiden lukumäärät [Martin ja Orr, 1990].

n	r	LOA	STO	CMP	ADD	MUL	DIV	ARR	FUN	RFUN	RAN	OTH
4	10K	85.28	26.90	12.32	15.91	1.00	0.00	15.35	0.00	0.91	1.00	0.91
10	10K	81.07	25.68	11.34	15.57	1.00	0.00	15.53	0.00	0.57	1.00	0.57
100	10K	76.74	24.47	10.38	15.17	1.00	0.00	15.83	0.00	0.18	1.00	0.18
1K	5K	75.52	24.14	10.11	15.04	1.00	0.00	15.95	0.00	0.06	1.00	0.06
10K	1K	75.16	24.04	10.04	15.01	1.00	0.00	15.98	0.00	0.02	1.00	0.02
100K	100	75.06	24.01	10.01	15.01	1.00	0.00	15.99	0.00	0.01	1.00	0.01

Taulukko 8.7. Koodausten generointi: operaatioiden lukumäärät [Atkinson ja Sack, 1992].

n	r	LOA	STO	CMP	ADD	MUL	DIV	ARR	FUN	RFUN	RAN	OTH
4	10K	95.93	28.02	7.73	23.11	0.00	1.91	25.11	0.00	0.00	0.75	0.00
10	10K	150.15	43.10	11.72	36.88	0.00	3.22	38.88	0.00	0.00	0.90	0.00

Taulukko 8.8. Koodausten generointi: operaatioiden lukumäärät [Johnsen, 1991].

n	r	LOA	STO	CMP	ADD	MUL	DIV	ARR	FUN	RFUN	RAN	OTH
4	10K	103.10	27.41	20.21	18.03	0.75	0.00	15.25	0.00	0.00	0.75	0.00
10	10K	105.20	27.34	20.15	19.26	0.90	0.00	16.30	0.00	0.00	0.90	0.00
100	10K	106.10	27.13	20.05	19.96	0.99	0.00	16.93	0.00	0.00	0.99	0.00
1K	5K	106.01	27.03	20.00	19.99	1.00	0.00	16.99	0.00	0.00	1.00	0.00
10K	1K	105.33	26.87	19.82	19.86	1.00	0.00	16.95	0.00	0.00	1.00	0.00
100K	100	105.95	26.99	19.99	19.99	1.00	0.00	17.00	0.00	0.00	1.00	0.00

Taulukko 8.9. Koodausten generointi: operaatioiden lukumäärät [Korsh, 1993].

n	r	LOA	STO	CMP	ADD	MUL	DIV	ARR	FUN	RFUN	RAN	OTH
4	10K	36.82	16.25	3.83	3.00	2.00	0.00	9.91	0.00	0.00	2.00	9.91
10	10K	36.14	15.50	3.89	3.00	2.00	0.00	9.87	0.00	0.00	2.00	9.87
100	10K	35.97	15.05	3.98	3.00	2.00	0.00	9.96	0.00	0.00	2.00	9.96
1K	5K	35.99	15.00	4.00	3.00	2.00	0.00	9.99	0.00	0.00	2.00	9.99
10K	1K	36.00	15.00	4.00	3.00	2.00	0.00	10.00	0.00	0.00	2.00	10.00
100K	100	36.00	15.00	4.00	3.00	2.00	0.00	10.00	0.00	0.00	2.00	10.00

Taulukko 8.10. Koodausten generointi: operaatioiden lukumäärät (Rémyn algoritmi).

8.3. Puiden konstruointi koodausten perusteella: operaatioiden lukumäärät

Seuraavaksi taulukoimme lopullista muotoa olevien binääripuiden konstruoinnin kustannukset, kun binääripuiden koodaukset on annettu (taulukot 8.11 - 8.15). Vertailukriteereinä käytettyjen operaatioiden lukumäärien laskentaperiaate on sama kuin edellisessä kohdassa. Suurin solmulukumäärä testeissä oli n=10000. Saatujen lukujen perusteella voimme tarvittaessa ekstrapoloida sarakkeen n=100000 luvut (poikkeus: taulukko 8.13).

n	r	LOA	STO	CMP	ADD	MUL	DIV	ARR	PTR	FUN	RFUN	RAN	OTH
4	10K	13.75	4.75	3.00	2.00	0.00	0.00	2.00	3.50	0.00	2.00	0.00	0.00
10	10K	13.90	4.90	3.00	2.00	0.00	0.00	2.00	3.80	0.00	2.00	0.00	0.00
100	10K	13.99	4.99	3.00	2.00	0.00	0.00	2.00	3.98	0.00	2.00	0.00	0.00
1K	5K	14.00	5.00	3.00	2.00	0.00	0.00	2.00	4.00	0.00	2.00	0.00	0.00
10K	1K	14.00	5.00	3.00	2.00	0.00	0.00	2.00	4.00	0.00	2.00	0.00	0.00

Taulukko 8.11. Lopullisten puiden konstruointi: tasapainotetut sulkujonot [Arnold and Sleep, 1980; Atkinson and Sack, 1992].

n	r	LOA	STO	CMP	ADD	MUL	DIV	ARR	PTR	FUN	RFUN	RAN	OTH
4	10K	9.75	4.25	2.50	0.75	0.00	0.00	0.75	1.50	0.00	1.00	0.00	0.75
10	10K	11.10	4.70	2.80	0.90	0.00	0.00	0.90	1.80	0.00	1.00	0.00	0.90
100	10K	11.91	4.97	2.98	0.99	0.00	0.00	0.99	1.98	0.00	1.00	0.00	0.99
1K	5K	11.99	5.00	3.00	1.00	0.00	0.00	1.00	2.00	0.00	1.00	0.00	1.00
10K	1K	12.00	5.00	3.00	1.00	0.00	0.00	1.00	2.00	0.00	1.00	0.00	1.00

Taulukko 8.12. Lopullisten puiden konstruointi: oikean etäisyyden menetelmä [Martin and Orr, 1990].

n	r	LOA	STO	CMP	ADD	MUL	DIV	ARR	PTR	FUN	RFUN	RAN	OTH
4	10K	9.75	3.88	2.50	1.13	0.00	0.00	0.75	1.50	0.00	1.00	0.00	0.00
10	10K	11.10	4.25	2.80	1.35	0.00	0.00	0.90	1.80	0.00	1.00	0.00	0.00

Taulukko 8.13. Lopullisten puiden konstruointi: oksastus [Johnsen, 1991].

n	r	LOA	STO	CMP	ADD	MUL	DIV	ARR	PTR	FUN	RFUN	RAN	OTH
4	10K	12.75	5.75	2.00	2.00	0.00	0.00	2.00	1.50	0.00	1.00	0.00	2.00
10	10K	12.90	5.90	2.00	2.00	0.00	0.00	2.00	1.80	0.00	1.00	0.00	2.00
100	10K	12.99	5.99	2.00	2.00	0.00	0.00	2.00	1.98	0.00	1.00	0.00	2.00
1K	5K	13.00	6.00	2.00	2.00	0.00	0.00	2.00	2.00	0.00	1.00	0.00	2.00
10K	1K	13.00	6.00	2.00	2.00	0.00	0.00	2.00	2.00	0.00	1.00	0.00	2.00

Taulukko 8.14. Lopullisten puiden konstruointi: esijärjestys-numeroparit [Korsh, 1993].

n	r	LOA	STO	CMP	ADD	MUL	DIV	ARR	PTR	FUN	RFUN	RAN	OTH
4	10K	23.25	8.75	6.75	0.00	0.00	0.00	4.50	4.00	0.00	2.25	0.00	9.00
10	10K	21.90	8.30	6.30	0.00	0.00	0.00	4.20	4.00	0.00	2.10	0.00	8.40
100	10K	21.09	8.03	6.03	0.00	0.00	0.00	4.02	4.00	0.00	2.01	0.00	8.04
1K	5K	21.01	8.00	6.00	0.00	0.00	0.00	4.00	4.00	0.00	2.00	0.00	8.00
10K	1K	21.00	8.00	6.00	0.00	0.00	0.00	4.00	4.00	0.00	2.00	0.00	8.00

Taulukko 8.15. Lopullisten puiden konstruointi: solmujen taulukointi (Rémyn algoritmi).

8.4. Yhteenlasketut operaatioiden lukumäärät

Tässä kohdassa taulukoimme koodausten generoinnin ja lopullisten puiden konstruoinnin yhteenlasketut yksikkökustannukset kunkin algoritmin osalta (taulukot 8.16 - 8.21). Sarakkeen ‘kd’ (vast. ‘puut’) kukin luku on vastaavan koodausten generointitaulukon (vast. puiden konstruointitaulukon) yhden vaakarivin lukujen summa. Tähdellä merkityssä sarakkeessa suluissa ilmoitettu luku on algoritmin kustannus, kun ‘periaatteessa tarpeettomat’ silmukat (ks. kohta 7.1.2) on karsittu.

Taulukoissa on listattu siis vain vertailukriteereinä käytettyjen operaatioiden lukumäärien summia kiinnittämättä huomiota siihen, kuinka raskaita eri operaatiot ovat toisiinsa nähden. Esimerkiksi yksi satunnaislukugeneraattorin kutsu on taulukon luvuissa samanarvoinen minkä tahansa muun operaation kanssa vaikka se koostuu useasta yksinkertaisemmasta operaatiosta. Lopullista muotoa olevan puun solmun luonnin kustannukset on jätetty pois, koska laskentatavasta johtuen tämä lukumäärä on aina täsmälleen yksi. Painokertoimet eri operaatioille asetetaan jäljempänä.

n	r	kd	kd*		puut		yht.	yht.*
4	10K	66.98	(66.98)	+	31.00	=	97.98	(97.98)
10	10K	76.13	(76.13)	+	31.60	=	107.73	(107.73)
100	10K	84.82	(84.82)	+	31.96	=	116.78	(116.78)
1K	5K	85.88	(85.88)	+	32.00	=	117.88	(117.88)
10K	1K	85.99	(85.99)	+	32.00	=	117.99	(117.99)
100K	100	86.00	(86.00)	+	32.00	=	118.00	(118.00)

Taulukko 8.16. Yhteenlasketut operaatioiden lukumäärät: Arnold ja Sleep [1980].

n	r	kd	kd*		puut		yht.	yht.*
4	10K	64.47	(57.97)	+	21.25	=	85.72	(79.22)
10	10K	81.75	(74.95)	+	24.10	=	105.85	(99.05)
100	10K	95.29	(88.31)	+	25.81	=	121.10	(114.12)
1K	5K	96.83	(89.83)	+	25.98	=	122.81	(115.81)
10K	1K	96.98	(89.98)	+	26.00	=	122.98	(115.98)
100K	100	97.00	(90.00)	+	26.00	=	123.00	(116.00)

Taulukko 8.17. Yhteenlasketut operaatioiden lukumäärät: Martin ja Orr [1990].

n	r	kd	kd*		puut		yht.	yht.*
4	10K	159.57	(127.82)	+	31.00	=	190.57	(158.82)
10	10K	152.34	(122.84)	+	31.60	=	183.94	(154.44)
100	10K	144.94	(116.79)	+	31.96	=	176.90	(148.75)
1K	5K	142.87	(114.85)	+	32.00	=	174.87	(146.85)
10K	1K	142.27	(114.27)	+	32.00	=	174.27	(146.27)
100K	100	142.10	(114.10)	+	32.00	=	174.10	(146.10)

Taulukko 8.18. Yhteenlasketut operaatioiden lukumäärät: Atkinson ja Sack [1992].

n	r	kd	kd*		puut		yht.	yht.*
4	10K	182.55	(176.05)	+	20.50	=	203.05	(196.55)
10	10K	284.84	(278.04)	+	23.20	=	308.04	(301.24)

Taulukko 8.19. Yhteenlasketut operaatioiden lukumäärät: Johnsen [1991].

n	r	kd	kd*		puut		yht.	yht.*
4	10K	185.49	(141.99)	+	29.00	=	214.49	(170.99)
10	10K	190.05	(147.45)	+	29.60	=	219.65	(177.05)
100	10K	192.14	(150.08)	+	29.96	=	222.10	(180.04)
1K	5K	192.02	(150.02)	+	30.00	=	222.02	(180.01)
10K	1K	190.83	(148.83)	+	30.00	=	220.83	(178.83)
100K	100	191.92	(149.92)	+	30.00	=	221.92	(179.92)

Taulukko 8.20. Yhteenlasketut operaatioiden lukumäärät: Korsh [1993].

n	r	kd	kd*		puut		yht.	yht.*
4	10K	83.72	(75.47)	+	58.50	=	142.22	(133.97)
10	10K	82.27	(74.77)	+	55.20	=	137.47	(129.97)
100	10K	81.92	(74.87)	+	53.22	=	135.14	(128.09)
1K	5K	81.98	(74.98)	+	53.02	=	135.00	(128.00)
10K	1K	82.00	(75.00)	+	53.00	=	135.00	(128.00)
100K	100	82.00	(75.00)	+	53.00	=	135.00	(128.00)

Taulukko 8.21. Yhteenlasketut operaatioiden lukumäärät: Rémyn algoritmi.

8.5. Yhteenlasketut painotetut kustannukset

Tässä kohdassa taulukoimme koodausten generoinnin ja lopullisten puiden konstruoinnin yhteenlasketut painotetut kustannukset kunkin algoritmin osalta (taulukot 8.23 - 8.28). Operaatioiden painokertoimet on listattu taulukossa 8.22. Mitään ‘oikeita’ operaatioiden painokertoimia ei ole olemassa, eikä valittujen kerrointen valintaa perustella. Muutettujen painokerrointen mukaiset taulukkoarvot on haluttaessa helppo laskea uudelleen taulukoiden 8.5 - 8.15 mukaisten operaatioiden lukumäärien perusteella. Sarake ‘vk’ (vakiokustannukset) sisältää yhden lopullista muotoa olevan puun solmun luonnin painotetut kustannukset. Koska kaikki kustannukset ilmoitetaan yhtä solmua kohti, yhtä taulukon vaakariviä kohti siis luodaan aina täsmälleen yksi solmu. Satunnaislukugeneraattorin painokerroin on laskettu sen suorittamien perus-operaatioiden painotettujen kustannusten perusteella. Tähdellä merkityt sarakkeet ilmaisevat algoritmien kustannukset, kun ‘periaatteessa tarpeettomat’ silmukat (ks. kohta 7.1.2) on karsittu.

LOA	STO	CMP	ADD	MUL	DIV	ARR	PTR	FUN	RFUN	RAN	OTH
1	1	1.5	1.5	2	2	4.5	2	6	10	32	2

Taulukko 8.22. Operaatioiden painokertoimet.

n	r	kd	kd*		puut		vk.		yht.	yht.*
4	10K	115.59	(115.59)	+	62.00	+	36.00	=	213.59	(213.59)
10	10K	142.87	(142.87)	+	62.90	+	36.00	=	241.77	(241.77)
100	10K	167.68	(167.68)	+	63.44	+	36.00	=	267.12	(267.12)
1K	5K	170.66	(170.66)	+	63.49	+	36.00	=	270.16	(270.16)
10K	1K	170.97	(170.97)	+	63.50	+	36.00	=	270.47	(270.47)
100K	100	171.00	(171.00)	+	63.50	+	36.00	=	270.50	(270.50)

Taulukko 8.23. Yhteenlasketut painotetut kustannukset: Arnold ja Sleep [1980].

n	r	kd	kd*		puut		vk.		yht.	yht.*
4	10K	106.77	(99.40)	+	36.75	+	36.00	=	179.52	(172.15)
10	10K	133.22	(125.47)	+	40.80	+	36.00	=	210.02	(202.27)
100	10K	153.04	(145.07)	+	43.23	+	36.00	=	232.27	(224.30)
1K	5K	155.25	(147.25)	+	43.47	+	36.00	=	234.72	(226.73)
10K	1K	155.48	(147.48)	+	43.50	+	36.00	=	234.97	(226.97)
100K	100	155.50	(147.50)	+	43.50	+	36.00	=	235.00	(227.00)

Taulukko 8.24. Yhteenlasketut painotetut kustannukset: Martin ja Orr [1990].

n	r	kd	kd*		puut		vk.		yht.	yht.*
4	10K	268.49	(232.36)	+	62.00	+	36.00	=	366.49	(330.36)
10	10K	257.86	(224.21)	+	62.90	+	36.00	=	356.76	(323.11)
100	10K	246.90	(214.74)	+	63.44	+	36.00	=	346.34	(314.18)
1K	5K	243.81	(211.79)	+	63.49	+	36.00	=	343.30	(311.29)
10K	1K	242.91	(210.91)	+	63.50	+	36.00	=	342.41	(310.41)
100K	100	242.64	(210.64)	+	63.50	+	36.00	=	342.14	(310.14)

Taulukko 8.25. Yhteenlasketut painotetut kustannukset: Atkinson ja Sack [1992].

n	r	kd	kd*		puut		vk.		yht.	yht.*
4	10K	311.00	(303.62)	+	35.44	+	36.00	=	382.44	(375.06)
10	10K	476.32	(468.57)	+	39.22	+	36.00	=	551.54	(543.79)

Taulukko 8.26. Yhteenlasketut painotetut kustannukset: Johnsen [1991].

n	r	kd	kd*		puut		vk.		yht.	yht.*
4	10K	281.97	(232.47)	+	50.50	+	36.00	=	368.47	(318.97)
10	10K	295.61	(247.01)	+	51.40	+	36.00	=	383.01	(334.41)
100	10K	303.07	(255.01)	+	51.94	+	36.00	=	391.01	(342.95)
1K	5K	303.45	(255.45)	+	51.99	+	36.00	=	391.45	(343.44)
10K	1K	302.00	(254.00)	+	52.00	+	36.00	=	390.00	(342.00)
100K	100	303.40	(255.40)	+	52.00	+	36.00	=	391.40	(343.40)

Taulukko 8.27. Yhteenlasketut painotetut kustannukset: Korsh [1993].

n	r	kd	kd*		puut		vk.		yht.	yht.*
4	10K	195.72	(186.35)	+	110.88	+	36.00	=	342.60	(333.22)
10	10K	194.14	(185.59)	+	104.35	+	36.00	=	334.49	(325.94)
100	10K	194.25	(186.19)	+	100.44	+	36.00	=	330.68	(322.63)
1K	5K	194.45	(186.45)	+	100.04	+	36.00	=	330.50	(322.49)
10K	1K	194.49	(186.49)	+	100.00	+	36.00	=	330.50	(322.50)
100K	100	194.50	(186.50)	+	100.00	+	36.00	=	330.50	(322.50)

Taulukko 8.28. Yhteenlasketut painotetut kustannukset: Rémyn algoritmi.

8.6. Asymptoottiset kustannukset

Taulukoihin 8.29 - 8.32 on koottu havaitut algoritmien kustannukset, kun n=100000. Kohtien 8.2 - 8.5 perusteella teemme päätelmiä siitä, edustavatko tässä kohdassa annetut luvut asymptoottista raja-arvoa vai eivät. Lisäksi tarkastelemme lukujen hajontaa eri ajokerroilla, algoritmien käyttämien lukujen suuruutta ja algoritmien tilankäyttöä.

Lopullisten puiden konstruointialgoritmien kohdalla asymptoottiset raja-arvot selvästikin löytyivät, joten seuraavassa tarkastelemme koodausten generointi-algoritmeja.

Algoritmien ARN, MAR sekä REM kohdalla asymptoottiset raja-arvot on likimäärin saavutettu: tapausten n=10000 ja n=100000 välinen ero kustannuksissa on mitätön.

Algoritmin ATK luvut näyttävät lähestyvän raja-arvoa, mutta jotta tällainen löytyisi, pitäisi puiden kokoa ja testisarjan pituutta suurentaa. Kun solmujen lukumäärät ovat n=4, 10, 100, 1000, 10000 ja 100000, koesarjan kustannusten (taulukko 8.18, sarake ‘kd’) suhteelliset muutokset n:n kasvaessa edelliseen n:n arvoon verrattuna olivat prosenteissa ilmaistuna 4.5, 4.9, 1.4, 0.4 ja 0.1. Suhteellisen eron muutoksen suppenemisvauhti ei ollut riittävän säännöllistä, jotta täsmällistä raja-arvoa näiden lukujen perusteella voitaisiin arvioida. Taulukoiden 8.18 ja 8.25 perusteella katsomme kuitenkin, että raja-arvo on ‘likimäärin’ löydetty. Joka tapauksessa algoritmin ATK kustannukset n:n kasvaessa lähestyvät raja-arvoa hitaammin kuin algoritmien ARN, MAR sekä REM kustannukset.

Algoritmin KOR tapauksessa havaittujen kustannusten muutos n:n funktiona ei ollut monotonista. Tämä saattaa johtua siitä, ettei testisarjojen pituus algoritmin KOR suhteen ollut riittävä. Toisaalta, algoritmin eri ajokerroilla esiintyvä operaatioiden lukumäärän keskihajonta on muita algoritmeja suurempi (taulukko 8.33). Havaittujen lukujen perusteella voitaneen kuitenkin otaksua, että asymptoottinen raja on likimäärin löydetty, mutta sen löytyminen edellyttää pitempiä koesarjoja ja suurempia solmulukumääriä kuin algoritmin ATK tapauksessa.

Koska algoritmin JOH tapauksessa suurin n:n arvo oli 10, luvut eivät edusta asymptoottista raja-arvoa. Luvuista on lisäksi jätetty pois esitaulukointivaiheen kustannukset, jotka ovat kvadraattisia.

	LOA	STO	CMP	ADD	MUL	DIV	ARR	FUN	RFUN	RAN	OTH
ARN	44.00	12.00	6.00	14.00	4.00	2.00	2.00	0.00	0.00	2.00	0.00
MAR	49.00	11.00	3.00	24.00	5.00	2.00	2.00	0.00	0.00	1.00	0.00
ATK	75.06	24.01	10.01	15.01	1.00	0.00	15.99	0.00	0.01	1.00	0.01
JOH	150.15	43.10	11.72	36.88	0.00	3.22	38.88	0.00	0.00	0.90	0.00
KOR	105.95	26.99	19.99	19.99	1.00	0.00	17.00	0.00	0.00	1.00	0.00
REM	36.00	15.00	4.00	3.00	2.00	0.00	10.00	0.00	0.00	2.00	10.00

Taulukko 8.29. Koodausten generointi: operaatioiden lukumäärät, kun n=100000 (poikkeus: JOH, n=10).

	LOA	STO	CMP	ADD	MUL	DIV	ARR	PTR	FUN	RFUN	RAN	OTH
ARN	14.00	5.00	3.00	2.00	0.00	0.00	2.00	4.00	0.00	2.00	0.00	0.00
MAR	12.00	5.00	3.00	1.00	0.00	0.00	1.00	2.00	0.00	1.00	0.00	1.00
ATK	14.00	5.00	3.00	2.00	0.00	0.00	2.00	4.00	0.00	2.00	0.00	0.00
JOH	11.10	4.25	2.80	1.35	0.00	0.00	0.90	1.80	0.00	1.00	0.00	0.00
KOR	13.00	6.00	2.00	2.00	0.00	0.00	2.00	2.00	0.00	1.00	0.00	2.00
REM	21.00	8.00	6.00	0.00	0.00	0.00	4.00	4.00	0.00	2.00	0.00	8.00

Taulukko 8.30. Lopullisten puiden konstruointi: operaatioiden lukumäärät, kun n=100000 (poikkeus: JOH, n=10).

	kd		kd*		puut		yht.		yht.*
ARN	86.00	(	86.00)	+	32.00	=	118.00	(	118.00)
MAR	97.00	(	90.00)	+	26.00	=	123.00	(	116.00)
ATK	142.10	(	114.10)	+	32.00	=	174.10	(	146.10)
JOH	284.84	(	278.04)	+	23.20	=	308.04	(	301.24)
KOR	191.92	(	149.92)	+	30.00	=	221.92	(	179.92)
REM	82.00	(	75.00)	+	53.00	=	135.00	(	128.00)

Taulukko 8.31. Yhteenlasketut operaatioiden lukumäärät, kun n=100000 (poikkeus: JOH, n=10).

	kd		kd*		puut		vk.		yht.		yht.*
ARN	171.00	(	171.00)	+	63.50	+	36.00	=	270.50	(	270.50)
MAR	155.50	(	147.50)	+	43.50	+	36.00	=	235.00	(	227.00)
ATK	242.64	(	210.64)	+	63.50	+	36.00	=	342.14	(	310.14)
JOH	476.32	(	468.57)	+	39.22	+	36.00	=	551.54	(	543.79)
KOR	303.40	(	255.40)	+	52.00	+	36.00	=	391.40	(	343.40)
REM	194.50	(	186.50)	+	100.00	+	36.00	=	330.50	(	322.50)

Taulukko 8.32. Yhteenlasketut painotetut kustannukset, kun n=100000 (poikkeus: JOH, n=10).

Taulukoiden 8.5 - 8.32 luvut on saatu yhdestä testisarjasta kaikkien algoritmien ja n:n arvojen osalta. Kustannusten hajontalukujen tutkimista varten testiajot ajettiin uudelleen koodausten generointialgoritmien osalta ja laskettiin operaatioiden yhteenlaskettujen lukumäärien keskiarvot (eli taulukoiden 8.16 - 8.21 sarakkeen ‘kd’ arvot) ja keskihajonnat eri algoritmeille (taulukko 8.33). Koska tutkittavien suureiden ‘oikeat’ jakaumat eivät ole tiedossa, laskennassa käytettiin otoksen perusteella lasketun keskihajonnan kaavaa [esim. Graham *et al.*, 1989, s. 278].

n	r	ARN	MAR	ATK	JOH	KOR	REM
4	10K	66.94/4.30	64.27/9.98	159.69/4.97	182.33/20.48	185.25/15.62	83.77/1.58
10	10K	76.12/3.44	81.75/6.06	152.29/3.55	284.99/46.65	189.93/15.38	82.28/0.85
100	10K	84.82/0.52	95.28/0.84	144.96/1.70	-	192.11/15.06	81.92/0.13
1K	5K	85.88/0.06	96.83/0.08	142.93/1.22	-	192.16/14.89	81.98/0.00
10K	1K	85.99/0.00	96.98/0.00	142.35/1.19	-	192.49/14.89	82.00/0.00
100K	100	86.00/0.00	97.00/0.00	142.08/1.19	-	192.27/14.55	82.00/0.00

Taulukko 8.33. Koodausten generointialgoritmien operaatioiden yhteenlaskettujen lukumäärien keskiarvot ja keskihajonnat.

Yhden puun generointiin käytettyjen operaatioiden yhteenlaskettujen lukumäärien keskihajonta oli algoritmeilla ARN, MAR ja REM olematon suurilla n:n arvoilla. Muiden algoritmien järjestys (pienemmästä suurempaan) oli ATK, KOR ja JOH. Poislukien algoritmi JOH, hajonta pieneni puun koon kasvaessa, mutta algoritmin KOR tapauksessa havaittu muutos oli hyvin hidasta.

Seuraavaksi taulukoimme koodausten generointialgoritmien käyttämien lukujen suuruuden n:n funktiona (taulukko 8.34). Satunnaislukugeneraattorin käyttämiä lukuja ei huomioida. Erottuu kaksi ryhmää: algoritmit ATK, KOR ja REM, joiden toiminnassa satunnaisuus toteutuu koodauksen sisältävän taulukon *positioiden* satunnaisessa valinnassa, sekä algoritmit ARN, MAR ja JOH, joissa satunnaisuus toteutuu valitsemalla satunnaisesti kunkin koodialkion *arvo*. Edellisessä ryhmässä käytettyjen lukujen suuruus on O(n), ja jälkimmäisessä ryhmässä suurempi. Algoritmin JOH luvut ovat eksponentiaalisia, mutta vastaavia satunnaisia binääripuun koodauksia on mahdollista generoida toisella tavalla lineaarisessa ajassa käyttäen lukuja, joiden suuruus on $O(n^2)$ [Mäkinen, 1999; ks. myös kohta 5.7].

ARN	MAR	ATK	JOH	KOR	REM
$O(n^2)$	$O(n^2)$	2n	exp	2n	2n

Taulukko 8.34. Koodausten generointialgoritmien käyttämien lukujen suuruus solmulukumäärän funktiona [Mäkinen, 1999].

Taulukossa 8.35 on esitetty koodausten generointialgoritmien toteutusten asymptoottinen tilankäyttö solmulukumäärän funktiona. Kun n on suuri, merkitsevää tilankäytön kannalta on ainoastaan n:stä riippuvien tietorakenteiden koko (ja tyyppi). Toteutuksissa pyrittiin optimoimaan ajankäyttöä ja sopeuttamaan tilankäyttö sen mukaan. Algoritmien ATK, JOH ja KOR toteutuksissa tarvittiin aputaulukoita. Ainakin algoritmin ATK toteutuksen tilankäyttöä olisi voitu parantaa, jos olisi tingitty ajoaikaisesta tehokkuudesta.

Algoritmeissa ATK, ARN ja KOR binääripuun koodaus on periaatteessa esitettävissä bittijonona. Algoritmin JOH esitaulukointivaiheessa rakennettavan taulukon tilavaatimus on $O(n^2)$. Algoritmi REM tarvitsee yhden taulukon, jonka koko on 2n+1, mutta jonka kukin alkio koostuu kolmesta taulukon indeksistä. Rekursiivisen algoritmin ATK ajoaikaisen kutsupinon korkeus on pahimmassa tapauksessa n. Jos pidämme binääriarvoisia taulukoita samanarvoisina kuin kokonaislukutaulukoita, tilankäytön suhteen algoritmien paremmuusjärjestys on MAR, ARN, KOR, REM, ATK, JOH.

ARN	MAR	ATK	JOH	KOR	REM
2n(bit)	n(int)	2n(bit) 4n(int)	$O(n^2)$	2n(bit) 2n(int)	6n(int)

Taulukko 8.35. Koodausten generointialgoritmien toteutusten tilankäyttö solmulukumäärän funktiona.

8.7. Keskusyksikköaika

Tässä kohdassa esitellään tulokset, jotka saatiin, kun mitattiin koodausten generointialgoritmien kuluttamaa keskusyksikköaikaa (taulukko 8.36). Testaus suoritettiin VAX 7000-830 osituskäyttöympäristössä. Koeajoja varten ohjelmien kaikki 'ylimääräinen' ohjelmakoodi poistettiin, siis myös lopullisten puiden konstruointimodulien kutsut. Mittaus käynnistettiin vasta juuri ennen kuin ohjelman suoritus eteni varsinaisen algoritmin suorituksen toistoja sisältävään silmukkaan, ja lopetettiin välittömästi ko. silmukan jälkeen.

Kustannusten laskentaperiaate on sama kuin aiemmissa kohdissa, ts. kustannukset lasketaan yhtä puun solmua kohti. Mitattu kokonaisaika siis jaettiin generoitujen puiden lukumäärällä r ja osamäärä jaettiin solmulukumäärällä n. On huomattava, että taulukon 8.36 luvut eivät tarkkaan ottaen ole yhteismitallisia, koska lopullista muotoa olevien binääripuiden konstruoinnin CPU-aikaa ei mitattu.

Ohjelmointiympäristön tarjoamat ajanlaskentapalvelut mittasivat CPU-aikaa yhden sadasosasekunnin paloina. Taulukon 8.36 luvut on jaettava luvulla 10^6, jotta saadaan selville, montako sekuntia CPU-aikaa keskimäärin kului yhtä puun solmua kohti. Saman koeajon eri toistokerroilla tai muutoin sellaisilla ajoilla, joiden odotettu lopullinen tulos oli sama (ts. n vakio, r vaihteli), yhteismitallisen ajan havaittu vaihtelu oli alle 2.5 prosenttia, paitsi algoritmilla KOR, jonka havaittu suurin vaihtelu oli noin 4.48 prosenttia. Puiden koodauksia generoitiin n:n arvoon 10^6 asti. Tällöin kuitenkin

yhteismitalliset luvut kasvoivat selvästi esim. n:n arvoon 10^5 verrattuna. Tämän selittää se, että ohjelman sisältämät taulukot mahtuivat keskusmuistiin, kun $n=10^5$, mutta eivät enää silloin, kun $n=10^6$, joten jälkimmäisessä tapauksessa aikakertymä aiheutui suureksi osaksi käyttöjärjestelmän suorittamasta virtuaalimuistin osien siirtelystä keskusmuistin ja sivutustiedoston välillä.

n	r	ARN	MAR	ATK	JOH	KOR	REM
4	500K	2.660	2.420	3.995	3.920	3.620	4.885
10	100K	3.690	2.940	3.750	5.680	3.870	4.780
100	50K	4.630	3.288	3.454	-	4.120	4.872
1K	10K	4.827	3.353	3.435	-	4.080	4.882
10K	1K	4.809	3.373	3.574	-	4.232	5.043
100K	100	4.785	3.361	3.988	-	4.640	5.674
1000K	10	4.797	-	7.593	-	7.881	10.978

Taulukko 8.36. Koodausten generointialgoritmien toteutusten suhteellisia keskusyksikköaikoja yhtä puun solmua kohti.

8.8. Koodausten generointialgoritmit: vertailua

Tässä kohdassa vertailemme tutkittavia koodausten generointialgoritmeja edellä esitettyjen taulukoiden perusteella. Ellei erikseen mainita, algoritmit luetellaan resurssien käytön suhteen paremmuusjärjestyksessä, ts. taloudellisimmat ensin. Algoritmien yhteydessä mainitut arvot ilmoittavat likimääräisesti, kuinka paljon algoritmi kuluttaa tarkasteltavana olevaa resurssia keskimäärin yhtä puun solmua kohti, kun puun koko kasvaa suureksi.

Ensiksi tarkastelemme koodausten generointialgoritmien yksittäisiä operaatioiden lukumääriä (ks. kohta 8.2).

Algoritmit MAR, ATK, JOH ja KOR kuluttavat yhden satunnaisluvun (operaatio RAN) yhtä puun solmua kohti, ARN ja REM kaksi. Tällä on merkitystä, koska käyttämämme satunnaislukugeneraattorin toteutus on raskas.

Taulukkoviittausten (operaatio ARR) suhteen algoritmit ARN (2 operaatiota) ja MAR (2) olivat selvästi parhaita, muiden järjestys on REM (10), ATK (16), KOR (17) ja JOH (>38). Huomioitava on kuitenkin myös se, että erilaisten taulukkoviittausten välillä vallitsee myös kvaliteettiero: tehdäänkö taulukkoviittauksia taulukon alkioihin ‘hajasaantityyppisesti’ vai käydäänkö taulukkoa läpi peräkkäin, jolloin taulukkoviittauksen voidaan olettaa olevan kevyempi. Asian merkitys tulee esille ainakin silloin, kun generoidaan niin suuria puita, etteivät niiden esitysmuodot mahdu kokonaisuudessaan keskusmuistiin. Tällöin ei-peräkkäisessä järjestyksessä läpikäytävät taulukot kuormittavat enemmän käyttöjärjestelmäpalveluita (ks. taulukko 8.36), ja

ohjelman suoritus etenee hitaammin. Algoritmit ARN ja MAR käyvät taulukkoa läpi ainoastaan peräkkäin; muut sisältävät kumpiakin tapoja. Tämä selittää myös sen, miksi algoritmin ARN suorituksen vertailukelpoisen keskusyksikköajan kasvuvauhti on pienempi kuin algoritmien ATK, KOR ja REM silloin, kun $n=10^6$.

Aritmeettisten operaatioiden (operaatiot ADD, MUL, DIV eli yhteen/vähennys-, kerto- ja jakolaskujen lukumäärät yhteenlaskettuna) suhteen järjestys oli REM (5), ATK (16), ARN (20), KOR (21), MAR (31) ja JOH (36). Lukuunottamatta algoritmia ARN kaikkien algoritmien yhteenlaskuista tarkemmin määrittelemätön osa on silmukoiden ohjausmuuttujien lisäysoperaatioita. Lisäksi suurin osa algoritmien yhteen- tai vähennyslaskuista on sellaisia, joissa muuttujan arvoa muutetaan yhdellä. Tällaisia operaatioita varten tietokoneiden käskyvalikoimassa on yleensä oma konekäsky (muuttujan ‘inkrementointi’), joten niiden suorituksen voidaan katsoa olevan tehokkaampia kuin ‘oikeiden’ yhteenlaskujen.

Algoritmin MAR suorituksessa, likimäärin n:n arvoista $9.5*10^5$ alkaen ilmeni laskentatarkkuuden menetys liian suurten lukujen vuoksi. Seurauksena oli ajoaikainen virhe (jako nollalla). Tämä todennäköisesti voitaisiin välttää paloittelemalla ja uudelleenryhmittelemällä algoritmin aritmeettisia lausekkeita.

Aritmeettisten ja loogisten vertailujen (operaatio CMP) osalta tulos oli MAR (3), REM (4), ARN (6), ATK (10), JOH (>11) ja KOR (20).

Muuttujaviittausten ja muuttujaan sijoitusten osalta (operaatiot LOA ja STO; lukumäärät laskettu yhteen) järjestys on REM (51), ARN (56), MAR (60), ATK (99), KOR (133) ja JOH (>190). Muuttujaan sijoitus on sama asia kuin lausekielen sijoituslause.

Erikseen laskettavia ei-rekursiivisia aliohjelmakutsuja (operaatio FUN) ei ole. Ne lasketaan joko erikseen (esim. satunnaislukugeneraattori) tai ne ovat kosmeettisia, jolloin ne jätetään huomiotta. Algoritmi ATK on ainoa, joka sisältää rekursiivisia kutsuja (operaatio RFUN), mutta niiden yhteismitallinen lukumäärä lähestyy n:n kasvaessa nollaa.

Algoritmista REM kirjattiin lisäksi tietueen kenttään viittaukset, joiden lukumäärä oli 10.

Kun edellämainitut operaatioiden lukumäärät lasketaan yhteen (kohta 8.4), järjestys on REM (82), ARN (86), MAR (97), ATK (142), KOR (192) ja JOH (285). Kun operaatioille asetetaan painot, joiden otaksumme likimäärin kuvaavan operaatioiden keskinäistä raskautta (kohta 8.5), järjestys muuttuu: MAR (155), ARN (171), REM

(195), ATK (243), KOR (303) ja JOH (476). Tämän selittää ennen muuta kaksi asiaa: algoritmi MAR käyttää yhden satunnaisluvun yhden puun solmun generoimiseen, algoritmit ARN ja REM kaksi. Satunnaislukugeneraattorin painokerroin on arvioitu sen suorittamien perusoperaatioiden perusteella, ja se on suuri (30). Algoritmi REM putoaa kolmanneksi, koska se suorittaa paljon taulukkoviittauksia (jotka lisäksi ovat hajasaantityyppisiä), kun taas MAR ja ARN suorittavat taulukkoviittauksia vähän (ja ne ovat peräkkäishakutyyppisiä). Algoritmi JOH oli viimeinen lähes kaikilla tavoilla mitattuna.

Suluissa olevat luvut taulukoissa 8.16 - 8.28 ilmoittavat kustannukset, kun 'periaatteessa tarpeettomien' eli 'yksinkertaisten' silmukoiden ohjauskustannukset on karsittu (ks. kohta 7.1.2). Algoritmi ARN ei tällaisia muodollisestikaan sisällä, eivätkä käytännössä myöskään algoritmit MAR ja REM, koska kahdessa viimeksimainitussa algoritmin koko runko on tällaisen silmukan sisällä. Sen sijaan algoritmin ATK vertailukriteerioperaatioista lukumäärällisesti noin 20 prosenttia on yksinkertaisten silmukoiden ohjausta, ja algoritmin KOR tapauksessa vastaava prosenttiosuus on noin 22. Kun kustannukset painotetaan, prosenttiosuudet ovat noin 13 ja 16.

Algoritmien tilankäyttöä, kustannusten hajontaa ja käytettyjen lukujen suuruutta tarkasteltiin kohdassa 8.6.

Keskusyksikköajan käytön suhteen tilanne on kaksijakoinen. Sen jälkeen, kun puiden kokoa kasvatettiin niin suureksi, etteivät ne mahtuneet kokonaisuudessaan keskusmuistiin, ohjelmien yhteismitalliset suoritusajat kasvoivat selvästi (taulukko 8.36). Jos yritämme minimoida ohjelmointiympäristöstä riippuvien tekijöiden vaikutusta tuloksiin (n:n tulisi kuitenkin olla mahdollisimman suuri), edustavimmat luvut löytynevät taulukon keskivaiheilta ($10^3 \leq n \leq 10^4$). Tällöin järjestys oli ($n=10^4$) MAR (3.37), ATK (3.57), KOR (4.23), ARN (4.81), REM (5.04) ja JOH (>5.6, n=10). Yllättävää oli algoritmin ARN keskinkertainen menestys, vaikka sen kosmeettiset aliohjelmakutsut poistettiin. Tämän selittänee algoritmin suorittama ahkera satunnaislukugeneraattorin käyttö. Toinen maininnanarvoinen seikka oli algoritmin ATK hyvä menestys, vaikka ko. algoritmi on rekursiivinen. Osan menestyksestä selittää se, että käytetty testausympäristö on rekursiivisten kutsujen kannalta tehokas.[2]

Kun tarkastelemme algoritmien suoritusaikoja kohdassa 8.7 kuvattujen

[2] Tehokkuus on demonstroitu seuraavasti: pöytätietokoneympäristössä suoritetun testin [Mäkinen, 1991c] mukaan binääripuiden luettelointialgoritmin rekursiivisen [Mäkinen, 1987] ja ei-rekursiivisen [Mäkinen, 1991c] version suoritusaikojen suhde oli likimäärin 12:1 ei-rekursiivisen version hyväksi. Kun sama testi toistettiin tässä tutkimuksessa käytetyssä ympäristössä, suhde oli noin 2:1.

muistinriittävyysongelmien esiintyessä ($n=10^6$), järjestys muuttuu toiseksi: ARN (4.80), ATK (7.59), KOR (7.88) ja REM (10.98). Tällöin jos algoritmi MAR olisi kyetty suorittamaan (muuttamalla aritmeettisten operaatioiden suoritusjärjestystä, ottamalla käyttöön erityyppisiä muuttujia tms.), taulukon 8.36 lukujen perusteella on todennäköistä, että algoritmi MAR olisi myös tässä listassa paras. Algoritmien ATK, KOR ja REM huonoon menestykseen vaikuttavat hajasaantityyppiset taulukot; algoritmin ATK tapauksessa vaikuttavana tekijänä on myös kutsupinon ajoaikaisen koon vaihtelu, jonka rekursiiviset kutsut aikaansaavat.

8.9. Puiden konstruointi koodausten perusteella: vertailua

Seuraavaksi vertailemme lopullista muotoa olevien puiden konstruointialgoritmeja koodausten perusteella. Näistä algoritmeista käytämme samoja kolmimerkkisiä lyhenteitä, joita olemme edellä käyttäneet merkitsemään koodausten generointialgoritmeja. Täten sekä ARN että ATK tarkoittavat algoritmia, joka rakentaa binääripuun tasapainotettujen sulkujonojen perusteella.

Algoritmit ARN/ATK sekä REM tekevät kaksi rekursiivista proseduurikutsua yhtä luotua puun solmua kohti, MAR, JOH ja KOR yhden. Kaikissa kutsuissa (poikkeus: algoritmi REM) on vain yksi välttämätön parametri, muut parametrit voidaan eliminoida esim. muuttamalla ne globaaleiksi muuttujiksi. Algoritmien REM ja KOR tarvitsema paikallisten muuttujien määrä jokaisella rekursiotasolla on 2 ja algoritmin MAR yksi; JOH sekä ARN/ATK eivät tarvitse paikallisia muuttujia. Paikallisten muuttujien hallintakustannukset on sisällytetty sarakkeen OTH kustannuksiksi. Algoritmin REM tapauksessa ko. sarake sisältää myös tietueen kenttään viittauksia. Poislukien algoritmi REM, aritmeettisten operaatioiden lukumäärissä ei ole suuria eroja, kuten ei myöskään osoitinmuuttujaviittausten, tavallisten muuttujaviittausten tai sijoituslauseiden määrissä.

Seuraavaksi tarkastelemme kokonaiskustannuksia. Painotettujen ja painottamattomien kustannusten osalta algoritmien järjestys on muuttumaton. Parhaat tulokset saivat algoritmit MAR ja JOH. Viimeksimainitun vertailuaineisto loppuu menetelmästä riippumattomasta syystä n:n arvoon 10. Kun em. kahden menetelmän kustannuksia vertailtiin n:n arvoilla 3...10, niiden kustannukset erosivat hyvin vähän toisistaan, emmekä näinollen määrittele niiden keskinäistä paremmuusjärjestystä.

Muiden algoritmien osalta merkittävää oli raskaimman resurssin (rekursiivinen

funktiokutsu, painokerroin 10) käyttöfrekvenssi, joka algoritmeilla ARN/ATK ja REM oli suurin. Algoritmin REM taulukkoviittaukset ovat lisäksi hajasaantityyppisiä (toisin kuin muilla algoritmeilla), joten sen taulukoidut luvut ovat itse asiassa liian optimistisia. Painotettuihin lukuihin lisäämme vakio edustaa kustannusta, joka aiheutuu puun solmulle muistista dynaamisesti varattavan tilan varaamisesta. Se on vakio (36), koska kaikki kustannukset lasketaan yhtä puun solmua kohti. Painotettuihin operaatioiden kustannuksiin perustuva algoritmien järjestys on täten JOH ja MAR (43.5+36), KOR (52+36), ARN/ATK (63.5+36) ja REM (100+36).

8.10. Yhdistetyt kustannukset: vertailua

Kohdissa 8.8 ja 8.9 vertailimme erilaisiin kriteereihin nojautuen koodausten generointialgoritmeja ja lopullista muotoa olevien puiden konstruointialgoritmeja koodausten perusteella. Kun erilaisia algoritmeja vertaillaan, tulosten voidaan katsoa olevan yhteismitallisia vasta kun em. kahden vaiheen tulokset yhdistetään, koska tällöin sekä lähtötilanne että lopputilanne ovat yhteismitallisia. Seuraavassa tarkastelemme yhdistettyjä tuloksia.

Kun laskemme yhteen koodausten generoinnin sekä puiden konstruoinnin kustannukset (taulukko 8.31), saamme operaatioiden lukumääriin eli painottamatomiin kustannuksiin perustuvaksi järjestykseksi ARN (118), MAR (123), REM (135), ATK (174), KOR (222) ja JOH (308). Parhaan tuloksen saanut koodausten generointialgoritmi REM ei kokonaisuutena ottaen ollut paras, koska soveltamiemme laskentakriteerien mukaan puun konstruointi koodauksen perusteella on tässä menetelmässä raskasta. Painotettujen kustannusten osalta (taulukko 8.32) järjestys oli MAR (235), ARN (271), REM (331), ATK (342), KOR (391) ja JOH (>550). Järjestys oli sama kuin koodausten generointialgoritmien tapauksessa.

9. Yhteenveto

Tässä tutkielmassa tutkimme kuuden satunnaisia binääripuita generoivan algoritmin keskinäistä tehokkuutta (luvut 7 ja 8). Lisäksi näytimme, miten satunnaisten binääripuiden generointialgoritmien toteutusten oikeellisuus voidaan tilastollisesti todentaa khiin neliö -testiä soveltamalla. Tämän menetelmän avulla myös löydettiin virhe kirjallisuudessa esitetyssä Rémyn algoritmin esimerkkitoteutuksessa: kaikki samankokoiset puut eivät olleet yhtä todennäköisiä.

Tässä tutkielmassa loimme yleiskatsauksen binääripuiden alkeisominaisuuksiin (luku 2), binääripuiden koodausmenetelmiin (luku 3), binääripuiden luettelointi-algoritmeihin liittyvään tutkimukseen (luku 4), yksityiskohtaisemmalla tasolla satunnaisten binääripuiden ja eräiden yleisempää muotoa olevien puiden generointi-algoritmeihin (luku 5) sekä eri todennäköisyysjakaumia noudattavien satunnaisten binääripuiden asymptoottisiin ominaisuuksiin (luku 6).

Tutkielman keskeinen osa oli kuuden satunnaisia binääripuita generoivan algoritmin suorituskyvyn vertailu. Melko tasavertaisten algoritmien kesken oli eri kriteerien perusteella mahdollista löytää useita erilaisia paremmuusjärjestyksiä. Taulukkoon 9.1 on koottu yhteenveto algoritmien paremmuusjärjestyksistä eri kriteerien suhteen. Termi ‘kd’ tarkoittaa koodausten generointialgoritmeja, ‘puut’ lopullista muotoa olevien puiden konstruointialgoritmeja, ‘kd+puut’ edellisten yhteenlaskettuja kustannuksia, ‘pain.’ painotettuja kustannuksia ja ‘ei-pain.’ painottamattomia kustannuksia, ‘CPU, m.’ tarkoittaa keskusyksikköaikaa silloin, kun muistia on riittävästi ja ‘CPU, ei-m.’ keskusyksikköaikaa silloin, kun ohjelmat eivät

kd, ei-pain.....:	REM	ARN	MAR	ATK	KOR	JOH
kd, pain.....:	MAR	ARN	REM	ATK	KOR	JOH
puut, ei-pain.....:	JOH/MAR	-	KOR	ARN/ATK	-	REM
puut, pain.....:	JOH/MAR	-	KOR	ARN/ATK	-	REM
kd+puut, ei-pain....:	ARN	MAR	REM	ATK	KOR	JOH
kd+puut, pain.....:	MAR	ARN	REM	ATK	KOR	JOH
hajonta, kd, ei-pain:	REM	ARN	MAR	ATK	KOR	JOH
käytetty tila, kd...:	MAR	ARN	KOR	REM	ATK	JOH
numeroiden suuruus..:	ATK/KOR/REM		-	ARN/MAR	-	JOH
CPU, kd, m.....:	MAR	ATK	KOR	ARN	REM	JOH
CPU, kd, ei-m.....:	ARN	ATK	KOR	REM	-	-

Taulukko 9.1. Algoritmien paremmuusjärjestyksiä eri kriteerien perusteella.

mahdu kokonaisuudessaan keskusmuistiin. Pientä kustannusten hajontaa pidämme parempana kuin suurta. Kauttaviivalla toisiinsa yhdistettyjen algoritmien paremmuus-järjestystä ei määritellä. Algoritmien nimien lyhennykset ja muut oletukset ja rajoitukset ovat samat kuin luvussa 8.

Operaatioiden lukumääriin perustuvalla kustannusten laskentamenetelmällä saatiin esille algoritmien ARN, MAR ja REM tarkat asymptoottiset suhteelliset kustannukset. Algoritmien ATK ja KOR tulokset olivat likimääräisiä algoritmien kustannusten suuremman hajonnan vuoksi. Tulokset olivat kuitenkin riittävän tarkkoja, jotta voitiin määritellä algoritmien keskinäinen järjestys. Tarkempien tulosten saamiseksi laskenta-kapasiteettia olisi pitänyt käyttää enemmän. Algoritmin JOH käyttämien lukujen eksponentiaalisuus teki käytetyssä testiympäristössä kustannusten asymptoottisen tarkastelun mahdottomaksi.

Algoritmit KOR ja ATK sisälsivät eniten 'periaatteessa tarpeettomien' silmukoiden ohjauskustannuksia. Tällaisten kustannusten huomiotta jättäminen ei kuitenkaan muuttanut algoritmien järjestystä.

Tutkittujen algoritmien lopullista ja yksiselitteistä paremmuusjärjestystä emme määrittele. Taulukossa 9.1 lueteltujen kriteerien välisen tärkeysjärjestyksen jätämme lukijan ratkaistavaksi. Yhtenä tämän tutkielman tuloksena voitaisiinkin pitää esimerkin antamista siitä, miten ‘näkökulmasidonnainen’ asia algoritmin O-lukua tarkemmalla tasolla tapahtuva tehokkuuden tutkiminen voi olla.

Algoritmien tehokkuuden tutkinnassa sovellettujen mittareiden suhteen jäi toivomisen varaa erityisesti taulukon alkioiden läpikäynnin kustannusten mittaamisessa, koska taulukon alkion saavuttamisen kustannus arvioitiin vakioksi riippumatta siitä, läpikäytiinkö taulukon alkiot peräkkäin vai satunnaisessa järjestyksessä. Tämän huomioonottaminen olisi oletettavasti rasittanut eniten Rémyn algoritmin toteutusta. Lisäksi operaatioiden raskautta kuvaavat painokertoimet olivat arvioita. Muutettujen painokerrointen mukaiset painotetut kustannukset ovat kuitenkin helposti laskettavissa uudelleen operaatioiden lukumääriä sisältävien taulukoiden perusteella.

Viiteluettelo

[Aho *et al.*, 1986] A.V. Aho, R. Sethi, J.D. Ullman, *Compilers - Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Aho and Ullman, 1972] A.V. Aho, J.D. Ullman, *The Theory of Parsing, Translation and Compiling, Vol. 1, Parsing*. Prentice-Hall, 1972.

[Aho and Ullman, 1995] A.V. Aho, J.D. Ullman, *Foundations of Computer Science, C Edition*. Computer Science Press, 1995.

[Alonso, 1994] L. Alonso, Uniform generation of a Motzkin word. *Theoret. Comput. Sci.* **134** (1994), 529-536.

[Alonso *et al.*, 1997] L. Alonso, J.L. Rémy, R. Schott, A linear-time algorithm for the generation of trees. *Algorithmica* **17** (1997), 162-182.

[Alonso *et al.*, 1997b] L. Alonso, J.L. Rémy, R. Schott, Uniform generation of a Schröder tree. *Inf. Proc. Letters* **64** (1997), 305-308.

[Alonso and Schott, 1995] L. Alonso, R. Schott, *Random Generation of Trees*. Kluwer Academic Publishers, 1995.

[Alonso and Schott, 1995b] L. Alonso, R. Schott, Random generation of colored trees. *Lecture Notes in Comp. Sci*. **911** (1995), 16-35.

[Arnold and Sleep, 1980] D.B. Arnold, M.R. Sleep, Uniform random generation of balanced parenthesis strings. *ACM Trans. Program. Lang. Syst.* **2** (1980), 122-128.

[Atkinson, 1993] M.D. Atkinson, Uniform generation of rooted ordered trees with prescribed degrees. *Comp. J.* **36** (1993), 593-594.

[Atkinson and Sack, 1992] M.D. Atkinson, J.-R. Sack, Generating binary trees at random. *Inf. Proc. Letters* **41** (1992), 21-23.

[Atkinson and Sack, 1994] M.D. Atkinson, J.-R. Sack, Uniform generation of forests of restricted height. *Inf. Proc. Letters* **50** (1994), 323-327.

[Baeza-Yates *et al.*, 1992] R. Baeza-Yates, R. Casas, J. Diaz, C. Martinez, On the average size of the intersection of binary trees. *SIAM J. Comput.* **21** (1992), 24-32.

[Bainbridge, 1992] J. Bainbridge, A heuristic method for generating large random expressions. *Inf. Proc. Letters* **44** (1992), 165-170.

[Bapiraju and Bapeswara Rao, 1994] V. Bapiraju, V.V. Bapeswara Rao, Enumeration of binary trees. *Inf. Proc. Letters* **51** (1994), 125-127.

[Barcucci *et al.*, 1994] E. Barcucci, R. Pinzani, R. Sprugnoli, The random generation of directed animals. *Theoret. Comput. Sci.* **127** (1994), 333-350.

[Berztiss, 1986] A. Berztiss, A taxonomy of tree traversals. *BIT* **26** (1986), 266-276.

[Beyer and Hedetniemi, 1980] T. Beyer, S.M. Hedetniemi, Constant time generation of rooted trees. *SIAM J. Comput*. **9** (1980), 706-712.

[Brinck and Foo, 1981] K. Brinck, N. Y. Foo, Analysis of algorithms on threaded trees. *Comp. J.* **24** (1981), 148-155.

[Carta, 1990] D.G. Carta, Two fast implementations of the "Minimal standard" random number generator. *Comm. ACM* **33** (1990), 87-88.

[Casas *et al.*, 1991] R. Casas, J. Diaz, C. Martinez, Statistics on random trees. *Lecture Notes in Comp. Sci*. **510** (1991), 186-203.

[Clarke and Cooke, 1988] G.M. Clarke, D. Cooke, *A Basic Course in Statistics*. Arnold, 1988.

[Dear, 1997] K. Dear, An Online Text in Introductory Statistics from the University of Newcastle. HTML-document, 1997. Available as *http://www.anu.edu.au/nceph/surfstat/surfstat-home/tables.html.*

[Dershowitz and Zaks, 1989] N. Dershowitz, S. Zaks, Patterns in trees. *Discrete Applied Math.* **25** (1989), 241-255.

[Devroye and Kruszewski, 1994] L. Devroye, P. Kruszewski. A note on the Horton-Strahler number for random trees. *Inf. Proc. Letters* **52** (1994), 155-159.

[Devroye and Kruszewski, 1995] L. Devroye, P. Kruszewski. The botanical beauty of random binary trees. In: *Proc. of the Symposium on Graph Drawing, GD '95, Lect. Notes in Comp. Sci.* **1027** (1995), 166-177.

[Devroye and Reed, 1995] L. Devroye, B. Reed, On the variance of the height of random binary search trees. *SIAM J. Comput*. **24** (1995), 1157-1162.

[Devroye and Robson, 1995] L. Devroye, J.M. Robson, On the generation of random binary search trees. *SIAM J. Comput.* **24** (1995), 1141-1156.

[Er, 1983] M.C. Er, A note on generating well-formed parenthesis strings lexicographically. *Comp. J*. **26** (1983), 205-207.

[Er, 1985] M.C. Er, Enumerating ordered trees lexicographically. *Comp. J*. **28** (1985), 538-542.

[Er, 1987] M.C. Er, On enumerating tree permutations in natural order. *Intern. J. Computer Math*. **22** (1987), 105-115.

[Er, 1989] M.C. Er, A new algorithm for generating binary trees using rotations. *Comp. J*. **32** (1989), 470-473.

[Even, 1979] S. Even, *Graph Algorithms*. Pitman, 1979.

[Feller, 1957] W. Feller, *An Introduction to Probability Theory and Its Applications, Vol. I, Second Edition*. John Wiley & Sons, 1957.

[Feller, 1966] W. Feller, *An Introduction to Probability Theory and Its Applications, Vol. II*. John Wiley & Sons, 1966.

[Flajolet and Odlyzko, 1982] P. Flajolet, A. Odlyzko, The average height of binary trees and other simple trees. *J. of Computer and System Sciences* **25** (1982), 171-213.

[Flajolet *et al.*, 1993] P. Flajolet, P. Zimmermann, B. Van Cutsem, A calculus of random generation. In: *Proc. of the 1st Annual European Symposium of Algorithms - ESA '93, Lecture Notes in Comp. Sci*. **726** (1993), Springer, 169-180.

[Flajolet *et al.*, 1994] P. Flajolet, P. Zimmermann, B. Van Cutsem, A calculus for the random generation of labelled combinatorial structures. *Theoret. Comput. Sci*. **132** (1994), 1-35.

[Furnas, 1984] G.W. Furnas, The generation of random, binary unordered trees. *J. of Classification* **1** (1984), 187-233.

[Gardner, 1976] M. Gardner, Mathematical games: Catalan numbers. *Scientific American* **234**, June 1976, 120-125.

[Gonnet and Baeza-Yates, 1991] G.H. Gonnet, R. Baeza-Yates, *Handbook of Algorithms and Data Structures*. Addison-Wesley, 1991.

[Goodrich and Tamassia, 1998] M.T. Goodrich, R. Tamassia, *Data Structures and Algorithms in Java*. John Wiley & Sons, 1998.

[Graham *et al.*, 1989] R.L. Graham, D.E. Knuth, O. Patashnik, *Concrete Mathematics*. Addison-Wesley, 1989.

[Gupta, 1991] D.K. Gupta, On the generation of P-sequences. *Intern. J. Computer Math*. **38** (1991), 31-35.

[Hickey and Cohen, 1983] T. Hickey, J. Cohen, Uniform random generation of strings in a context-free language. *SIAM J. Comput.* **12** (1983), 645-655.

[Hikita, 1983] T. Hikita, Listing and counting subtrees of equal size of a binary tree. *Inf. Proc. Letters* **17** (1983), 225-229.

[Hille, 1985] R.F. Hille, Binary trees and permutations. *Austr. Comp. J.* **17** (1985), 85-87.

[Horowitz and Sahni, 1987] E. Horowitz, S. Sahni, *Fundamentals of Data Structures in Pascal*. Computer Science Press, 1987.

[Iba, 1996] H. Iba, Random tree generation for genetic programming, In: *Late Breaking Papers at the Genetic Programming Conference at Stanford University,* 1996, 75-82.

[Itkonen, 1988] T. Itkonen, *Kieliopas, 4. painos.* 1988.

[Jerrum *et al.*, 1986] M.R. Jerrum, L.G. Valiant, V.V. Vazirani, Random generation of combinatorial structures from a uniform distribution. *Theoret. Comput. Sci*. **43** (1986), 169-188.

[Johnsen, 1991] B. Johnsen, Generating binary trees with uniform probability. *BIT* **31** (1991), 15-31.

[Joichi *et al.*, 1980] J.T. Joichi, D.E. White, S.G. Williamson, Combinatorial Gray codes. *SIAM J. Comput.* **9** (1980), 130-141.

[Kemp, 1998] R. Kemp, Generating words lexicographically: An average case analysis. *Acta Inf*. **35** (1998), 17-89.

[Knott, 1977] G.D. Knott, A numbering system for binary trees. *Comm. ACM* **20** (1977), 113-115.

[Knuth, 1969] D.E. Knuth, *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms.* Addison-Wesley, 1969.

[Knuth, 1973a] D.E. Knuth, *The Art of Computer Programming, Vol. 1, Fundamental Algorithms, Second Edition.* Addison-Wesley, 1973.

[Knuth, 1973b] D.E. Knuth, *The Art of Computer Programming, Vol. 3, Sorting and Searching*. Addison-Wesley, 1973.

[Korsh, 1993] J.F. Korsh, Counting and randomly generating binary trees. *Inf. Proc. Letters* **45** (1993), 291-294.

[Korsh, 1994] J.F. Korsh, Loopless generation of k-ary tree sequences. *Inf. Proc. Letters* **52** (1994), 243-247.

[Korsh and Lipschutz, 1998] J.F. Korsh, S. Lipschutz, Shifts and loopless generation of k-ary trees. *Inf. Proc. Letters* **65** (1998), 235-240.

[Lee *et al*., 1986] C.C. Lee, D.T. Lee, C.K. Wong, Generating binary trees of bounded height. *Acta Inf*. **23** (1986), 529-544.

[Lucas, 1987] J. Lucas, The rotation graph of binary trees is Hamiltonian. *J. of Algorithms* **8** (1987), 503-535.

[Lucas *et al*., 1993] J.M. Lucas, D. Roelants van Baronaigien, F. Ruskey, On rotations and the generation of binary trees. *J. of Algorithms* **15** (1993) 343-366.

[Mahmoud, 1986] H.M. Mahmoud, The expected distribution of degrees in random binary search trees. *Comp. J.* **29** (1986), 36-37.

[Mahmoud, 1995] H.M. Mahmoud, The joint distribution of the three types of nodes in uniform binary trees. *Algorithmica* **13** (1995), 313-323.

[Mairson, 1994] H.G. Mairson, Generating words in a context-free language uniformly at random. *Inf. Proc. Letters* **49** (1994), 95-99.

[Martin, 1991] H.W. Martin, The uniform selection of ordered k-ary trees having n nodes. Unpublished manuscript.

[Martin and Orr, 1990] H.W. Martin, B.O. Orr, A random binary tree generator. *Proc. ACM 17th Annual Computer Science Conference*, 1990, 33-38.

[Mäkinen, 1987] E. Mäkinen, Left distance binary tree representations. *BIT* **27** (1987), 163-169.

[Mäkinen, 1991] E. Mäkinen, A survey on binary tree codings. *Comp. J.* **34** (1991), 438-443.

[Mäkinen, 1991b] E. Mäkinen, A note on graftings, rotations and distances in binary trees. *EATCS Bull.* **46** (1991), 146-148.

[Mäkinen, 1991c] E. Mäkinen, Efficient generation of rotational-admissible codewords for binary trees. *Comp. J.* **34** (1991), 379.

[Mäkinen, 1998] E. Mäkinen, Binary tree code words as context-free languages. *Comp. J.* **41** (1998), 422-424.

[Mäkinen, 1999] E. Mäkinen, Generating random binary trees - a survey. *Inf. Sciences.* **115** (1999), 123-136.

[Mäkinen and Siltaneva, 2000] E. Mäkinen, J. Siltaneva, A note on Rémy's algorithm for generating random binary trees. University of Tampere, Dept. of Computer and Information Sciences, Report **A-2000-2**, January 2000. Also available from *http://www.cs.uta.fi/reports/r2000.html*.

[Nijenhuis and Wilf, 1975] A. Nijenhuis, H.S. Wilf, *Combinatorial Algorithms*. Academic Press, 1975.

[Pallo, 1986] J.M. Pallo, Enumerating, ranking and unranking binary trees. *Comp. J.* **29** (1986), 171-175.

[Pallo and Racca, 1985] J.M. Pallo, R. Racca, A note on generating binary trees in A-order and B-order. *Intern. J. Computer Math.* **18** (1985), 27-39.

[Park and Miller, 1988] S.K. Park, K.W. Miller, Random number generators: good ones are hard to find. *Comm. ACM* **31** (1988), 1192-1201.

[Proskurowski, 1980] A. Proskurowski, On the generation of binary trees. *JACM* **27** (1980), 1-2.

[Proskurowski and Ruskey, 1985] A. Proskurowski, F. Ruskey, Binary tree Gray codes. *J. of Algorithms* **6** (1985), 225-238.

[Quiroz, 1989] A.J. Quiroz, Fast random generation of binary, t-ary and other types of trees. *J. of Classification* **6** (1989), 223-231.

[Reingold *et al.*, 1977] E.M. Reingold, J. Nievergelt, N. Deo, *Combinatorial Algorithms: Theory and Practice.* Prentice-Hall, 1977.

[Roelants van Baronaigien, 1991] D. Roelants van Baronaigien, A loopless algorithm for generating binary tree sequences. *Inf. Proc. Letters* **39** (1991), 189-194.

[Roelants van Baronaigien and Ruskey, 1988] D. Roelants van Baronaigien, F. Ruskey, Generating t-ary trees in A-order. *Inf. Proc. Letters* **27** (1988), 205-213.

[Rosen, 1991] K.H. Rosen, *Discrete Mathematics and Its Applications, 2nd Ed*. McGraw-Hill, 1991.

[Rotem, 1975] D.Rotem, On a correspondence between binary trees and a certain type of permutation. *Inf. Proc. Letters* **4** (1975), 58-61.

[Rotem and Varol, 1978] D.Rotem, Y.L. Varol, Generation of binary trees from ballot sequences. *JACM* **25** (1978), 396-404.

[Ruskey, 1978] F. Ruskey, Generating t-ary trees lexicographically. *SIAM J. Comput*. **7** (1978), 424-439.

[Ruskey and Hu, 1977] F. Ruskey, T.C. Hu, Generating binary trees lexicographically. *SIAM J. Comput*. **6** (1977), 745-758.

[Ruskey and Proskurowski, 1990] F. Ruskey, A. Proskurowski, Generating binary trees by transpositions. *J. of Algorithms* **11** (1990), 68-84.

[Räihä, 1988] K.-J. Räihä, *Tietorakenteet, luvut 5-9*. Tampereen yliopisto, Tietojenkäsittelyopin laitos, 1988.

[Sedgewick and Flajolet, 1996] R. Sedgewick, P. Flajolet, *An Introduction to the Analysis of Algorithms.* Addison-Wesley, 1996.

[Semba, 1981] I. Semba, Generation of all the balanced parenthesis strings in lexicographical order. *Inf. Proc. Letters* **12** (1981), 188-192.

[Skarbek, 1988] W. Skarbek, Generating ordered trees. *Theoret. Comput. Sci.* **57** (1988), 153-159.

[Smith, 1989] J.D. Smith, *Design and Analysis of Algorithms.* PWS-KENT Publishing Company, 1989.

[Solomon and Finkel, 1980] M. Solomon, R.A. Finkel, A note on enumerating binary trees. *JACM* **27** (1980), 3-5.

[Sprugnoli, 1992] R. Sprugnoli, The generation of binary trees as a numerical problem. *JACM* **39** (1992), 317-327.

[Trojanowski, 1978] A.E. Trojanowski, Ranking and listing algorithms for k-ary trees. *SIAM J. Comput*. **7** (1978), 492-509.

[Uusi Sivistyssanakirja, 1992] A. Aikio, R. Vornanen, *Uusi Sivistyssanakirja, 11. painos*. Otava, 1992.

[Vajnovszki, 1998] V. Vajnovszki, On the loopless generation of binary tree sequences. *Inf. Proc. Letters* **68** (1998), 113-117.

[Viennot, 1990] X.G. Viennot, Trees everywhere. In: *Proc. of the C.A.A.P. '90, Lecture Notes in Comp. Sci*. **431** (1990), 18-41.

[Walsh, 1998] T.R. Walsh, Generation of well-formed parenthesis strings in constant worst-case time. *J. of Algorithms* **29** (1998), 165-173.

[Weiss, 1994] M.A. Weiss, *Data Structures and Algorithm Analysis in C++*. The Benjamin/Cummings Publishing Company, 1994.

[Wilf, 1981] H.S. Wilf, The uniform selection of free trees. *J. of Algorithms* **2** (1981), 204-207.

[Wirth, 1976] N. Wirth, *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.

[Xiang *et al.*, 1997] L. Xiang, C. Tang, K. Ushijima, Grammar-oriented enumeration of binary trees. *Comp. J.* **40** (1997), 278-291.

[Zaks, 1980] S. Zaks, Lexicographic generation of ordered trees. *Theoret. Comp. Sci*. **10** (1980), 63-82.

[Zaks and Richards, 1979] S. Zaks, D. Richards, Generating trees and other combinatorial objects lexicographically. *SIAM J. Comput.* **8** (1979), 73-81.

[Zerling, 1985] D. Zerling, Generating binary trees using rotations. *JACM* **32** (1985), 694-701.

Liite: testeissä käytetyt algoritmien toteutukset

```
//
// generoidaan satunnainen tasapainotettu sulkujono
// [Arnold and Sleep, 1980]
//

const char Open = 1, Close = 0;

void ArnoldSleep_Open( int &r, int &k, int word[], int &ind )
{
  ind = ind + 1;
  word[ind] = Open;
  r = r + 1;                          __t(add,3); __t(arr,1);
  k = k - 1;                          __t(loa,8); __t(sto,4);
}

void ArnoldSleep_Close( int &r, int &k, int word[], int &ind )
{
  ind = ind + 1;
  word[ind] = Close;
  r = r - 1;                          __t(add,3); __t(arr,1);
  k = k - 1;                          __t(loa,8); __t(sto,4);
}

void ArnoldSleep_ProbClose( int &r, int &k, double &prob )
{
  prob = double( r * ( r + k + 2 ) ) / double( 2 * k * ( r + 1 ) );
                                      __t(mul,2); __t(add,4); __t(divi,1);
                                      __t(loa,8); __t(sto,1);
}


void main()
{
  const maxN...                       // puun max koko
  int   n,                            // solmujen lukumäärä
        word[2*maxN+1];               // koodisana 1..2n

  int   k, r, ind=0;
  double r1, r2;

  // alustukset...

  k = 2*n;
  r = 0;
  ind = 0;                            __t(loa,4); __t(add,1); __t(sto,3);

  while (r != k)
  {                                   __t(loa,4); __t(cmp,2); //while+if
    if (r == 0)
      ArnoldSleep_Open(r, k, word, ind);      //ei kutsukust.
    else
      { r1 = randomGen();             __t(ran,1); __t(sto,1);

        ArnoldSleep_ProbClose(r, k, r2);      //ei kutsukust.

                                      __t(loa,2); __t(cmp,1);
        if (r1 < r2)
          ArnoldSleep_Close(r, k, word, ind); //ei kutsukust.
        else
          ArnoldSleep_Open(r, k, word, ind);  //ei kutsukust.
      }
  }                                   __t(loa,2); __t(cmp,1); //while (r!=k)

  while (k != 0)
  {                                   __t(loa,2); __t(cmp,1); //while (k!=0)
    ArnoldSleep_Close(r, k, word, ind);       //ei kutsukust.
  }                                   __t(loa,2); __t(cmp,1); //while (k!=0)

  // muodostetaan koodauksen perusteella binääripuu;
  // treeBuild(...);

}
```

```
=========================================================================
//
// generoidaan satunnainen tasapainotettu sulkujono
// [Atkinson and Sack, 1992]
//

int   word[4*maxN+1],     // koodisana (pituus 2n) rakentuu tähän
      alkuPos, loppuPos;  // muuttujan word koodauksen alku ja loppupositiot

//
// koodausta ei voi rakentaa suoraan muuttujaan temp seur. syystä:
// kun u ei ole tasapainotettu, temp pitää uudelleenjärjestää;
// tämä aikaansaa sen, että toteutus ei testien mukaan ole lineaarinen;
// koodaus pitää siis rakentaa muuttujaan word keskeltä, ja
// koodaus voi kasvaa sekä eteen- että taaksepäin, jolloin
// uudelleenjärjestämistä ei tarvitse tehdä;
// seuraus: toteutus on lineaarinen
//

void AtkinsonSackGen(int temp[], int tempLen)
{
  int prefixLen;
  static int i, partialSum;
                                    __t(rfun,1);__t(oth,1);
                                    __t(loa,2); __t(cmp,1);
  if (tempLen > 0)
  {
    partialSum = prefixLen = 0;     __t(loa,2); __t(sto,2);
    do
    { if (temp[prefixLen++] == 1)
        partialSum++;
      else
        partialSum--;               __t(loa,7); __t(add,2); __t(arr,1);
                                    __t(cmp,1); __t(sto,2);
                                    __t(loa,2); __t(cmp,1);
    } while (partialSum != 0);

    // käsittele v
    AtkinsonSackGen(temp+prefixLen,
                    tempLen-prefixLen);__t(loa,4); __t(add,2);

                                    __t(loa,3); __t(cmp,1); // ehto
    // tasapainotettu alkuosa u jos temp[0]==1
    if (temp[0])
      {                             __t(loa,1); __t(add,1); // for i=...
                                    __t(forReal,prefixLen);
        // kopioidaan u wordiin v:n eteen
        for (i=prefixLen-1; i>=0; i--)
        { word[--alkuPos] = temp[i];  __t(arr,2); __t(add,1);
                                    __t(loa,5); __t(sto,2);
        }
      }
    else
      { // kopioidaan wordiin alkusulku v:n eteen
        word[--alkuPos]  = 1;       __t(arr,1+1); __t(add,1+1);
                                    __t(loa,4+4); __t(sto,2+2);
        // kopioidaan loppusulku v:n jälkeen
        word[++loppuPos] = 0;
                                    __t(forReal,prefixLen-2);
                                    __t(loa,2); __t(add,1); __t(sto,1);
        // käsittele t*, missä u=)t( ja vie tulos v:n jälkeen
        for (i=1; i<=prefixLen-2; i++)
        {
          word[++loppuPos] = 1-temp[i];__t(arr,2); __t(add,2);
                                     __t(loa,6); __t(sto,2);
        }
      }
  }
}


void main()
{
  int   n, _2n, ind, r, apu;
  int   temp[2*maxN];               //apumuuttuja
  double r1;
```

```
  // alustukset...
  _2n = 2*n;
                                    __t(forKnown,_2n); __t(loa,4*_2n);
                                    __t(arr,2*_2n); __t(sto,2*_2n);
  for (ind=0; ind<_2n; ind++)
    { temp[ind] = 0;
      word[ind] = ind;
    }
                                    __t(forKnown,n); __t(ran, n);
                                    __t(loa,11*n); __t(sto,5*n);
                                    __t(arr, 4*n); __t(add,2*n);
                                    __t(mul, 1*n);
  // generoidaan muuttujaan temp satunnainen 1-0 -jono, missä molempia
  // sama määrä
  for (ind=0; ind<n; ind++)
    { r1 = randomGen();
      r = r1 * (_2n-ind) + ind;
      apu = word[ind];
      word[ind] = word[r];
      word[r] = apu;
    }
                                    __t(forKnown,n); __t(loa,3*n);
                                    __t(arr,2*n); __t(sto,n);
  for (ind=0; ind<n; ind++)
    temp[word[ind]] = 1;


  // generoidaan tasapainotettu sulkujono muuttujaan word, jonka alku-
  // ja loppupositiot asetetaan osoittamaan tyhjää merkkijonoa
  //
  alkuPos  = _2n+1;                 __t(loa,5); __t(add,1); __t(sto,2);
  loppuPos = _2n;
  AtkinsonSackGen( temp, _2n );


  // muodostetaan koodauksen perusteella binääripuu;
  // treeBuild(...);

}
=============================================================================
//
// generoidaan satunnainen oikean etäisyyden koodisana
// [Martin and Orr, 1990]
//

void main()
{
  int   i, j, k, n,
        x[maxN+1];                  // koodisana 1..n
  double random, sum, p, q;

  // alustukset...

  x[1] = 0;                         __t(loa,2);  __t(arr,1); __t(sto,1);
                                    __t(forKnown,n-1); //n-1 kuten vakio
  for (j=1; j<=n-1; j++)
  {
    i = x[j];                       __t(loa,4);  __t(sto,2);
    k = i+1;                        __t(add,1);  __t(arr,1);

    p = double( (i+3)*(n-j) ) / double( (i+2)*(2*n-2*j+i+1) );
                                    __t(divi,1); __t(add,8); __t(mul,2);
                                    __t(loa,12); __t(sto,1);
    sum = p;
    random = 1.0-randomGen();       __t(ran,1);
                                    __t(loa,2);  __t(add,1); __t(sto,2);

    while (random > sum)
    {                               __t(loa,2);  __t(cmp,1); //while

      q = double( (k+1)*(n-j+k+1) ) / double( (k+2)*(2*n-2*j+k-1) );
                                    __t(add,10); __t(mul,2); __t(divi,1);
                                    __t(loa,14); __t(sto,1);
      p = q*p;
      sum = sum+p;                  __t(loa,6);  __t(sto,3);
      k = k-1;                      __t(mul,1);  __t(add,2);
    }                               __t(loa,2);  __t(cmp,1); //while
```

```
    x[j+1] = k;                       __t(loa,3);  __t(add,1);
                                      __t(arr,1);  __t(sto,1);
  }

  // muodostetaan koodauksen perusteella binääripuu;
  // treeBuild(...);

}
=============================================================================
//
// generoidaan satunnainen binääripuun koodisana (oksastus)
// [Johnsen, 1991]
//

void main()
{
  int   i, ind, n, a, b, m, j,
        nArr[maxN+1],
        x[maxN+1];                    // koodisana 1..n-1

  float random, pdf[maxN+1], cdf[maxN+2];
  int   L[maxN+2][maxN];
  int   jInd, mInd;

  // alustukset...

  //
  // rakennetaan todennäköisyystaulukko L
  //
  L[2][0] = 1;

  for (jInd=1; jInd<=n-1; jInd++)
    { L[2][jInd] = 4*L[2][jInd-1] - 6*L[2][jInd-1]/(jInd+2);
    }

  for (jInd=0; jInd<=n-2; jInd++)
    { L[3][jInd] = L[2][jInd+1] - L[2][jInd];
    }

  for (mInd=4; mInd<=n+1; mInd++)
    {
      for (jInd=0; jInd<=n-mInd+1; jInd++)
        { L[mInd][jInd] = L[mInd-1][jInd+1] - L[mInd-2][jInd+1];
        }
    }

  //
  // generoidaan koodaus
  //
  x[0] = 0;
  nArr[0] = 1;                        __t(loa,4); __t(arr,2); __t(sto,2);

                                      __t(forKnown,n-1); //n-1 vakio
  for (i=1; i<=n-1; i++)
  {
    nArr[i] = nArr[i-1] + 1 - x[i-1]; __t(add,4); __t(arr,3);
                                      __t(loa,8); __t(sto,1);

                                      __t(forReal,nArr[i]);
                                      __t(loa,3); __t(add,1); __t(arr,1);
                                      __t(sto,1); //simuloi x=nArr[i]-1
    for (ind=0; ind<=nArr[i]-1; ind++)
    {
      m = nArr[i]+1-ind;
      j = n-1-i;
      a = L[m][j];                    __t(arr,3); __t(add,4);
                                      __t(loa,10);__t(sto,3);

      m = nArr[i];
      j = n-i;
      b = L[m][j];                    __t(arr,3); __t(add,1); __t(divi,1);
      pdf[ind] =  float(a) / float(b);__t(loa,10);__t(sto,4);
    }

    cdf[0] = 0.0;                     __t(loa,1); __t(arr,1); __t(sto,1);
```

```
                                    __t(forReal,nArr[i]); //x=nArr[i]
                                    __t(loa,2); __t(arr,1); __t(sto,1);
    for (ind=1; ind<=nArr[i]; ind++)
      { cdf[ind] = cdf[ind-1] + pdf[ind-1];   __t(arr,3); __t(add,2);
                                              __t(loa,7); __t(sto,1);
      }

    random = randomGen();           __t(ran,1); __t(loa,3);
    ind = nArr[i] - 1;              __t(arr,1); __t(add,1); __t(sto,2);

                                    __t(loa,3); __t(arr,1); __t(cmp,1);
    while (random < cdf[ind])
      { ind = ind - 1;              __t(loa,5); __t(arr,1); __t(cmp,1);
                                    __t(add,1); __t(sto,1);
      }

    x[i] = ind;                     __t(loa,2); __t(arr,1); __t(sto,1);
  }

  // muodostetaan koodauksen perusteella binääripuu;
  // treeBuild(...);

}
=========================================================================
//
// generoidaan satunnainen binääripuun koodisana (esijärjestys-numeroparit)
// [Korsh, 1993]
//

main()
{
  int   n, _2n, ind, ind2, r, apu,
        diff, alkupos, alkupospit;
  int   word[2*maxN+1],             // koodisana 1..2n
        temp[2*maxN+1];
  double r1;

  // alustukset...
  _2n = 2*n;
                                    __t(forKnown,_2n); __t(arr,2*_2n);
                                    __t(loa,4*_2n);    __t(sto,2*_2n);
  for (ind=1; ind<= _2n; ind++)
    { temp[ind] = 0;
      word[ind] = ind; }
                                    __t(forKnown,n-1); //n-1 kuten vakio
                                    __t(ran,  n-1);
                                    __t(loa,12*(n-1)); __t(sto,5*(n-1));
                                    __t(arr, 4*(n-1)); __t(add,3*(n-1));
                                    __t(mul, 1*(n-1));
  // generoidaan muuttujaan temp satunnainen 1-0 -jono, missä ykkösiä n-1
  // ja nollia n+1
  for (ind=1; ind<=n-1; ind++)
    { r1 = randomGen();
      r = r1 * (_2n-ind+1) + ind;
      apu = word[ind];
      word[ind] = word[r];
      word[r] = apu;
    }
                                    __t(forKnown,n-1);
                                    __t(loa,3*(n-1));  __t(arr,2*(n-1));
                                    __t(sto,n-1);
  for (ind=1; ind<=n-1; ind++)
    { temp[word[ind]] = 1; }

  // generoidaan tasapainotettu sulkujono
                                    __t(loa,4); __t(sto,4);
  alkupos = 1; alkupospit = 0; diff = 0; ind = 0;

  while (alkupospit < _2n)
  {                                 __t(loa,2); __t(cmp,1);
    alkupospit = alkupospit + 1;    __t(loa,2); __t(add,1); __t(sto,1);

                                    __t(loa,2); __t(cmp,1);
    if (ind == _2n)
      { ind = 1;                    __t(loa,1); __t(sto,1);
      }
    else
```

```
      { ind++;                          __t(loa,2); __t(add,1); __t(sto,1);
      }
                                        __t(loa,5); __t(arr,1); __t(cmp,1);
    if (temp[ind] == 0) diff++; else diff--;      __t(add,1); __t(sto,1);

                                        __t(loa,2); __t(cmp,1);
    if (diff < 2)
      ;
    else
      {                                 __t(loa,2); __t(cmp,1);
        if (alkupospit >= _2n)
          ;
        else
          { diff = 0;     // 'siirrä' alkuosa loppuun, aloita alusta
            alkupos = ind+1;
            alkupospit = 0;             __t(loa,4); __t(add,1); __t(sto,3);
          }
      }
  }
  __t(loa,2); __t(cmp,1);  // while

  //
  // suoritetaan validin koodauksen aikaansaava rotaatio
  //
  ind2 = alkupos;                       __t(loa,1); __t(sto,1);
                                        __t(forKnown,_2n);
                                        __t(loa,3*_2n); __t(arr,2*_2n); //rivi 1
                                        __t(sto,1*_2n);                 //rivi 1
                                        __t(loa,4*_2n); __t(add,1*_2n); //rivi 2
                                        __t(cmp,1*_2n);                 //rivi 2
  for (ind=1; ind<= _2n; ind++)
    {
      word[ind] = temp[ind2];           //rivi 1
                                        //rivi 2 if-rivi
      if (++ind2 > _2n)
        { ind2 = 1;                     __t(loa,1); __t(sto,1);
        }
    }

  // muodostetaan koodauksen perusteella binääripuu;
  // treeBuild(...);

}
=========================================================================
//
// generoidaan satunnainen binääripuun koodaus (solmujen taulukointi)
// [Mäkinen and Siltaneva, 2000]
//

#define NIL  -1

struct Node
  { int parent, left_child, right_child; };
struct Node tree[2*maxN+1];


void growing_tree(int n, int &root)
{ int i, hit, vasen, parent;
  double r1;

  // puu, jossa vain lehtisolmu
  root = 0;
  tree[0].left_child  = NIL;
  tree[0].right_child = NIL;            __t(loa,7); __t(arr,3);
  tree[0].parent = NIL;                 __t(oth,3); __t(sto,4);

                                        __t(forKnown,n);
  for (i=1; i<2*n+1; i=i+2)             // for-lause: n kertaa; 2n+1 vakio
  {                                     __t(ran,2);
                                        __t(loa,4); __t(mul,2); __t(sto,4);
    r1 = randomGen();
    hit = r1 * i;     // sat.luku väliltä [0,i-1], puun solmun valinta
    r1 = randomGen();
    vasen = r1 * 2;   // satunnaisluku 0/1 eli vasen/oikea

    // asetetaan uuden sisäsolmun vanhempisolmun tiedot
    parent = tree[hit].parent;          __t(loa,2); __t(sto,1);
```

```
                                    __t(arr,1); __t(oth,1);

                                    __t(loa,2); __t(cmp,1);
    if (parent == NIL)
      { root = i;                   __t(loa,1); __t(sto,1);
      }
    else if (tree[parent].left_child == hit)
      { tree[parent].left_child  = i;
                                    __t(loa,3); __t(cmp,1); //ehto
                                    __t(arr,1); __t(oth,1);
                                    __t(loa,2); __t(oth,1); //lohko
                                    __t(arr,1); __t(sto,1);
      }
    else
      { tree[parent].right_child = i;
                                    __t(loa,3); __t(cmp,1); //ehto
                                    __t(arr,1); __t(oth,1);
                                    __t(loa,2); __t(oth,1); //lohko
                                    __t(arr,1); __t(sto,1);
      }
                                    __t(loa,2); __t(arr,1);
                                    __t(oth,1); __t(sto,1);
    // asetetaan uuden sisäsolmun tiedot
    tree[i].parent = parent;
                                    __t(loa,7); __t(cmp,1); __t(add,1);
                                    __t(arr,2); __t(oth,2); __t(sto,2);
    if (vasen)    // if (vasen != 0)
      { tree[i].left_child  = i+1;
        tree[i].right_child = hit; }
    else
      { tree[i].left_child  = hit;
        tree[i].right_child = i+1; }

    // asetetaan solmun v tiedot
    tree[hit].parent      = i;        __t(loa,2); __t(arr,1);
                                      __t(oth,1); __t(sto,1);
    // asetetaan uuden lehtisolmun tiedot
    tree[i+1].left_child  = NIL;      //simuloi apumuuttujaa x=i+1
    tree[i+1].right_child = NIL;      __t(loa,8); __t(add,1);
    tree[i+1].parent      = i;        __t(arr,3); __t(oth,3); __t(sto,4);
  }
}
=========================================================================
//
// [Alonso and Schott, 1995, Appendix 2, Alg. 2.1]
// Generation of binary trees: Remy's algorithm

// construction of a growing binary tree with n internal nodes

// modifies the position of two leaves, does not modify the tree built
void change_leaves (int a, int b)
{ int parenta, parentb;
  parenta = tree[a].parent;
  parentb = tree[b].parent;

  if (tree[parenta].right_child == a)
    tree[parenta].right_child = b;
  else
    tree[parenta].left_child = b;

  tree[a].parent = parentb;

  if (tree[parentb].right_child == b)
    tree[parentb].right_child = a;
  else
    tree[parentb].left_child = a;

  tree[b].parent = parenta;
}


void growing_tree (int n)
{ int i, number, tmp;
  double r1;

  // build a tree of size 1
  tree[0].left_child  = 1;
```

```
  tree[0].right_child = 2;
  tree[0].num = 1;

  tree[1].parent = tree[2].parent = 0;
  tree[1].right_child = tree[1].left_child = -1;
  tree[2].right_child = tree[2].left_child = -1;

  // the internal nodes will be in the boxes [0,i-1[ of the tree's array
  // the leaves in boxes [i,2*i[ of the tree's array

  for (i=2; i<=n; i++)
  {
    number = random(i); // random number of [0,i[
    // the leaf is in the box number+i-1
    change_leaves(i-1, number+i-1);

    tree[i-1].right_child = 2*i-1;
    tree[i-1].left_child  = 2*i;
    tree[i-1].num = i;

    tree[2*i-1].parent = tree[2*i].parent = i-1;
    tree[2*i-1].right_child = tree[2*i-1].left_child = -1;
    tree[2*i].right_child = tree[2*i].left_child = -1;
  }
}
=========================================================================
//
// lopullista muotoa olevan puun konstruointi koodauksen perusteella
//

int   word[2*maxN+1],                 // binääripuun koodaus
      wordInd,                        // indeksi
      n,                              // puun solmujen lkm
      _2n;                            // 2n

//
// puun solmu
//
struct tnode
{
  struct tnode *left, *right;         // osoittimet lapsisolmuihin
};

=========================================================================
//
// tasapainotetut sulkujonot: puun konstruointi
//

struct tnode *treeBuild_Paren(struct tnode *p)
{                                     __t(rfun,1);
                                      __t(loa,5); __t(add,1); __t(arr,1);
                                      __t(sto,1); __t(cmp,1);
  if (word[++wordInd] == 1)
    { p = tnodeNew();                 //kutsun kust. lasketaan muualla
      p->left  = treeBuild_Paren(p->left);
                                      __t(loa,3); __t(sto,2);
                                      __t(oso,2); __t(cmp,1);
      if (wordInd < _2n)
        { p->right = treeBuild_Paren(p->right);
                                      __t(oso,2); __t(loa,1); __t(sto,1);
        }
    }
  return(p);
}
=========================================================================
//
// oikean etäisyyden koodaus: puun konstruointi
//

struct tnode *treeBuild_MartinOrr(struct tnode *p, int &koodi, const int &n)
{                                     __t(rfun,1);
  p = tnodeNew();                     //kutsun kust. lasketaan muualla
                                      __t(sto,1); __t(loa,2); __t(cmp,1);
  if (wordInd >= n)
    { koodi = -1;                     __t(loa,1); __t(sto,1);
    }
  else
```

```
    { int kd = koodi;                 __t(oth,1);
      koodi = word[++wordInd];        __t(add,1); __t(arr,1); __t(cmp,2);
                                      __t(loa,9); __t(sto,3);
      if (koodi > kd)
        { p->left = treeBuild_MartinOrr(p->left, koodi, n);
                                      __t(loa,1); __t(oso,2); __t(sto,1);
        }
      if (koodi == kd)
        { p->right = treeBuild_MartinOrr(p->right, koodi, n);
                                      __t(loa,1); __t(oso,2); __t(sto,1);
        }
    }
  return(p);
}
=========================================================================
//
// oksastus: puun konstruointi
//

struct tnode *treeBuild_Johnsen(struct tnode *p, int &tlNo, const int &n)
{                                     __t(rfun,1);
  p = tnodeNew();                     //kutsun kust. lasketaan muualla
                                      __t(sto,1); __t(loa,2); __t(cmp,1);
  if (wordInd >= n)
    { tlNo = -1;                      __t(loa,1); __t(sto,1);
    }
  else
    { tlNo = word[++wordInd];         __t(add,1); __t(arr,1); __t(sto,2);
                                      __t(cmp,1); __t(loa,6);
      if (tlNo == 0)
        { p->left  = treeBuild_Johnsen(p->left, tlNo, n);
                                      __t(loa,1); __t(oso,2); __t(sto,1);
        }

                                      __t(loa,2); __t(cmp,1);
      if (tlNo == 1)
        { p->right = treeBuild_Johnsen(p->right, tlNo, n);
                                      __t(loa,1); __t(oso,2); __t(sto,1);
        }
      else
        { tlNo = tlNo - 1;            __t(loa,2); __t(add,1); __t(sto,1);
        }
    }
  return(p);
}
=========================================================================
//
// esijärjestys-numeroparit: puun konstruointi
//

struct tnode *treeBuild_Korsh(struct tnode *p)
{
  int left, right;                    __t(rfun,1);__t(oth,2);
  left  = word[++wordInd];            __t(add,2); __t(arr,2);
  right = word[++wordInd];            __t(loa,8); __t(sto,4);

  p = tnodeNew();                     //kutsun kust. lasketaan muualla
                                      __t(sto,1);

                                      __t(loa,4); __t(cmp,2);
  if (left  == 1)
    { p->left  = treeBuild_Korsh(p->left); __t(sto,1); __t(oso,2); __t(loa,1);
    }
  if (right == 1)
    { p->right = treeBuild_Korsh(p->right);__t(sto,1); __t(oso,2); __t(loa,1);
    }

  return(p);
}
=========================================================================
//
// solmujen taulukointi: puun konstruointi
//

struct tnode *treeBuild_Remy(struct tnode *p, int indeksi, const int n)
{ int left, right;
                                      __t(rfun,1);__t(oth,2);
```

```
  left  = tree[indeksi].left_child;   __t(arr,2); __t(loa,4);
  right = tree[indeksi].right_child;  __t(oth,2); __t(sto,2);

                                      __t(loa,4); __t(cmp,3);
  if (left == -1 && right == -1)
    { p = NULL;                       __t(loa,1); __t(sto,1);
    }
  else
    { p = tnodeNew();                 //kutsun kust. lasketaan muualla
      p->left  = treeBuild_Remy(p->left,  left,  n);
      p->right = treeBuild_Remy(p->right, right, n);
                                      __t(loa,4); __t(oso,4); __t(sto,3);
    }
  return(p);
}
=============================================================================
```